Army Research Laboratory

Testing, Selection, and Implementation of Random Number Generators

by Joseph C. Collins

ARL-TR-4498 July 2008

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5068

ARL-TR-4498 **July 2008**

Testing, Selection, and Implementation of Random Number Generators

Joseph C. Collins
Survivability/Lethality Analysis Directorate, ARL

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* July 2008	**2. REPORT TYPE** Final	**3. DATES COVERED (From - To)** June 2007–November 2007

4. TITLE AND SUBTITLE Testing, Selection, and Implementation of Random Number Generators	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Joseph C. Collins	**5d. PROJECT NUMBER** 7LB5D1
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: AMSRD-ARL-SL-BD Aberdeen Proving Ground, MD 21005-5068	**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-4498
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
An exhaustive evaluation of state-of-the-art random number generators with several well-known suites of tests provides the basis for selection of suitable random number generators for use in stochastic simulations.

Implementation details for the selected algorithms include the synthesis of a virtually unlimited number of extremely long-period statistically independent random streams in an efficient and transparent manner.

15. SUBJECT TERMS
random number, F2-linear, independence, testing, selection

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Joseph C. Collins
a. REPORT UNCLASSIFIED	**b. ABSTRACT** UNCLASSIFIED	**c. THIS PAGE** UNCLASSIFIED	UL	82	**19b. TELEPHONE NUMBER** *(Include area code)* 410-278-6832

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

INTENTIONALLY LEFT BLANK.

1. Random Number Generators

The random number generators (RNGs) evaluated in this report fall into a few main categories. The RNG algorithms are listed in appendix A. Initialization (seeding) algorithms are not presented.

The first category includes variations on linear congruential generator (LCG) algorithms.

Library functions, denoted `lib.rand`, `lib.rand_r`, and `lib.random`, are generally not portable. These are executed from library calls, and the implementations depend on platform. The extent of standardization (*1*) is only that: "The `rand` function computes a sequence of pseudo-random integers in the range 0 to RAND_MAX" and "The value of the RAND_MAX macro shall be at least 32767."

In an effort to be specific, `cyg.rand` is from Cygwin C library source code, `glib.rand_r` is from GNU C library source code, and `Cstd.rand1` is from the essential reference (*2*). On some platforms, these may coincide with library functions, although this is not required by the standard.

The MUVES-S2 RNG `rnrand` is a variant of `Cstd.rand1`.

The standard Java RNG `java32` is from Java library source.

Multiplicative linear congruential generator (MLCG) algorithms include the Park-Miller minimal standard RNG with Bays-Durham shuffle (*3*) `pm3bds`, which is the original RNG implementation in MUVES 3, and `cyg.rand_r`, taken from Cygwin C library source code.

The next two groups are based on shift register (SR) implementation.

RNGs based on the linear feedback shift register (LFSR) include Tausworthe (*4*, *5*) RNGs, the 32-bit `taus088` and `taus113`, and the 64-bit `taus258` (tested as `taus258x` and `taus258x32`).

Variations on generalized feedback shift register (GFSR) algorithms include the Ziff (*6*) RNG `gfsr`, a four-tap GFSR algorithm, the Mersenne Twister (*7*, *8*) 32-bit version `mt` and 64-bit version `mt64` (tested as `mtx` and `mtx32`), and the well-equidistributed long-period linear (*9*) RNG `well1024a`.

RNGs due to Marsaglia are multiply with carry (MWC) variations `cmwc4096`, `mwc`, `mwcx`, exclusive-or (XOR) Shift (*10*) `xor128`, and keep it simple stupid `kiss`, a combination of LFSR, LCG, and MWC elements.

2. Testing and Selection

Major test suites for RNGs are TestU01 (*11*), the NIST suite (*12*), and the Diehard suites DH1 and DH2 (*13*). The Diehard suites implement many classical RNG tests, the NIST suite is geared to cryptographic security, and TestU01 is extremely diverse, implementing classical tests, cryptographic tests, new tests proposed in the literature, and original tests.

All RNGs were tested with the complete DH1 and DH2 suites and NIST suite. All RNGs except for some of the library functions and algorithms were subjected to the major test batteries from TestU01.

Since all suites use 32-bit integers, the 64-bit algorithms were used to create 32-bit output either by taking the upper 32 bits in the case of `mtx` and `taus258x` or by alternating upper and lower 32-bit segments in the case of `mtx32` and `taus258x32`.

DH1 implements 24 tests and DH2 implements 26 tests. Each test produces a single p-value p. We regard that test as failed if $p < 0.005$, indicating departure from uniformity.

The NIST suite implements 189 tests and reports p's. A test is considered failed if $p < 0.0001$ or $p > 0.9999$.

The TestU0101 suite implements 773 tests and reports p's. A test is considered failed if $p < 0.001$ or $p > 0.999$.

The p-value selection criteria for the various test suites were chosen to produce a few failures in the best cases. Setting the criteria too low (closer to zero) would exhibit no failures, and setting the criteria too high would fail everything.

Table 1 tallies the failure counts for each RNG by test suite. Detailed test results are available on the local intranet (*14*) or by request from the author.

SR (LFSR and GFSR) RNGs are expected to fail Linear Complexity and Lempel-Ziv Compression tests. This causes concern for cryptographic use but is not relevant for our applications. In fact, the features of these algorithms exploited in section 3 imply such failures.

Additionally, LFSR (e.g., Tausworthe) RNGs are known to fail Matrix Rank tests. Here, each Tausworthe RNG fails eight TestU01 Matrix Rank tests. This may cause problems when simulating large binary matrices. Curiously, `well1024a`, which is supposed to avoid some of the problems of the Mersenne Twisters, fails four TestU01 Matrix Rank tests. The Mersenne Twisters pass all TestU01 Matrix Rank tests.

The NIST suite also tests Matrix Rank. Marginal failure for `mt32x` is the only NIST Matrix Rank test failure among the LFSR and GFSR RNGs.

Table 1. RNG test suite failure counts.

RNG			Test Suite				
Category	**Name**	**(size)**	**DH1**	**DH2**	**NIST**	**TestU01**	**TestU01[a]**
LCG	`cyg.rand`	(32)	1	1	7	17	16
	`glib.rand_r`	(31)	16	20	147	-	-
	`lib.rand`	(31)	13	15	148	-	-
	`lib.random`	(31)	13	15	149	-	-
	`lib.rand_r`	(31)	15	18	146	-	-
	`Cstd.rand1`	(31)	18	23	148	-	-
	`rnrand`	(15)	17	22	187	404	375
	`java32`	(32)	4	6	9	111	109
MLCG	`pm3bds`	(31)	5	10	149	157	154
	`cyg.rand_r`	(31)	15	18	146	-	-
LFSR	`taus088`	(32)	0	0	4	15	1
	`taus113`	(32)	0	0	5	16	2
	`taus258x`	(64)	0	0	6	12	0
	`taus258x32`	(64)	0	1	6	13	1
GFSR	`gfsr`	(32)	0	0	6	4	0
	`mt`	(32)	1	1	7	6	2
	`mtx`	(64)	1	0	2	4	0
	`mtx32`	(64)	1	0	9	5	1
	`well1024a`	(32)	0	0	11	9	1
MWC	`cmwc4096`	(32)	0	1	3	1	1
	`mwc`	(32)	0	1	8	2	2
	`mwcx`	(32)	0	0	8	2	2
XOR Shift	`xor128`	(32)	0	0	9	30	16
Combined	`kiss`	(32)	0	0	5	2	2

[a] Linear Complexity, Lempel-Ziv Compression, and Matrix Rank test failures excluded.

The Mersenne Twister is widely accepted by the community; in fact, `mt` is the core RNG implementation in the Python language and is a default RNG in Mathematica, PHP, and GLib.

The Tausworthe RNGs are efficient and can be extremely fast.

Based on these considerations and the test results, the author recommends use of the 32-bit Mersenne Twister `mt` as a primary RNG and the four-SR Tausworthe `taus113` as secondary. If 64-bit algorithms are required, the 64-bit Mersenne Twister `mt64` and five-SR Tausworthe `taus258` algorithms should be implemented.

These RNGs share a common structure that facilitates the implementation of statistically independent RNGs as shown in the following section.

3. $\mathbf{F_2}$-Linear RNGs

3.1 Motivation

A wide class of fast long-period RNGs are based on linear recurrence modulo 2. This class includes the LFSR Tausworthe and the GFSR Mersenne Twister family RNGs. The RNGs considered here have approximate periods p ranging from $2^{88} \sim 3 \cdot 10^{26}$ to $2^{19937} \sim 4 \cdot 10^{6001}$.

A necessary feature of RNG implementation is the availability of statistically independent RNGs. The creation of algorithmically independent RNGs is trivial. One simply allocates separate state information for each instance of the RNG and assures each RNG modifies only its own state.

However, this does not assure statistical independence. Consider that any RNG generates a single output sequence, and selecting the initial state of the RNG (called seeding) chooses a particular starting place in the sequence. Two instances are considered to be independent if their sequences do not overlap during execution of the program using the RNGs.

To obtain two maximally independent instances of an RNG, one could choose two starting points separated by $p/2$. One way to do this is to seed the first RNG, start the second RNG at the same point, and execute the second instance $p/2$ times to obtain its start state. The simplest Tausworthe RNG has period $p \sim 2^{88}$. One would initialize the second instance by executing it 2^{87} times. If the RNG executes 1 billion or $10^9 \sim 2^{30}$ times/second, this would take about 2^{57} s. Since there are about 2^{25} s in a year, the RNGs would be ready to use in about $2^{32} \sim 4 \cdot 10^9$ years. That's 4 billion years. The other Tausworthe RNGs to consider have periods $\sim 2^{113}$ and 2^{258}. The Mersenne Twister has $p \sim 2^{19937} - 1$. Initializing it in this way would take $> 2 \cdot 10^{5991}$ years. This is not practical.

Fortunately, the technology shown here can get this down to microseconds for the Tausworthe RNGs and a few milliseconds for the Mersenne Twister. The methods are due to L‘Ecuyer et al. (*15, 16*).

3.2 $\mathbf{F_2}$-Linear RNG Algorithms

All the RNGs mentioned previously work in essentially the same way. The theory is based on arithmetic in the finite field with two elements, $F_2 = \{0,1\}$. The binary operations of addition and multiplication in F_2 are characterized by

$$0+0=1+1=0\,, \quad \text{and} \quad 0+1=1+0=1 \tag{1}$$

and

$$0\times 0=1\times 0=0\times 1=0\,, \quad \text{and} \quad 1\times 1=1\,. \tag{2}$$

Since 1 is its own additive inverse, $a = -a$ in F_2, and subtraction is the same as addition. Bitwise “exclusive or”, denoted ˆ, is addition, so $a + b = a\,\hat{}\,b$. Bitwise “and”, denoted &, is multiplication, so $a \times b = a\&b$.

State information for the RNG is held in x, an element of the w-dimensional coordinate vector space F_2^w. In simple cases, x is an unsigned integer, and $w = 32$. The RNG is implemented by recursive linear transformation on the state

$$x_{n+1} = Bx_n\,, \quad \text{where} \quad n = 0, 1, 2, \ldots \tag{3}$$

to give the RNG sequence $x_0, x_1, x_2, \ldots$. Multiple steps are realized through repeated application of the transformation, symbolized by powers of B as $x_n = B^n x_0$ or, in general, $x_{n+i} = B^n x_i$. The identity transformation is $B^0 = I$.

All the polynomials we consider are elements of $F_2[z]$, the ring of polynomials in indeterminate z with coefficients in F_2. The transformation B has a characteristic polynomial (CP) of degree $k \leqslant w$,

$$C(z) = \det(B - zI) = z^k + c_{k-1}z^{k-1} + \cdots + c_1 z + c_0\,, \tag{4}$$

which satisfies $C(B) = 0$ according to the Cayley-Hamilton theorem (any linear transformation is a zero of its own CP). Therefore B^k can be expressed as a combination of lower-power terms,

$$B^k = c_{k-1}B^{k-1} + \cdots + c_1 B + c_0 I\,. \tag{5}$$

To compute some future state x_d directly from the initial state x_0,

$$x_d = B^d x_0\,, \tag{6}$$

apply the division algorithm to the polynomial z^d to obtain

$$z^d = Q(z) \cdot C(z) + J(z)\,, \tag{7}$$

where $J = 0$ or $\deg J < \deg C$. We write

$$J(z) = z^d \bmod C(z) = j_{k-1}z^{k-1} + j_{k-2}z^{k-2} + \cdots + j_2 z^2 + j_1 z + j_0\,. \tag{8}$$

Since $C(B) = 0$, we have $B^d = J(B)$, and the transformation to state d is an F_2-linear combination of the transformations $B^{k-1}, \cdots, B^0 = I$, with $j_i \in \{0, 1\}$,

$$B^d = j_{k-1}B^{k-1} + \cdots + j_0 B^0\,, \tag{9}$$

and state x_d is the same F_2-linear combination of the k initial states $x_i = B^i x_0$,

$$x_d = j_{k-1}x_{k-1} + \cdots + j_0 x_0\,, \tag{10}$$

where arithmetic in the coordinate vector space F_2^w yields the final result, the jump state x_d.

The CP C is only calculated once per RNG. The jump polynomial (JP) J must be calculated for each d. But this can be stored and applied to any RNG state to obtain the state d iterations forward. The Tausworthe generators use three, four, or five parallel 32-bit or 64-bit words and thus have $w \leqslant 64$ and require fewer than 64 states to perform the calculation with no more than five words of state information (this is accomplished in microseconds). The Mersenne Twisters use 624 32-bit words or 312 64-bit words for 19,968 bits of state information. These require fewer than 21,000 states for calculation of any future state and executes in milliseconds.

3.3 An Example

Consider the first subgenerator of taus088. The algorithm is denoted `gen`, and calling `gen()` updates x to Bx. In generic C/C++ code, the RNG algorithm follows.

```
unsigned x = 0x12340f00;
unsigned gen ( void ) {
  x = ( ( x & 0xfffffffe ) << 12 ) ^ ( ( ( x << 13 ) ^ x ) >> 19 );
  return x;
}
```

The initial (seed) state of x is given. Executing `gen()` advances x to the next state and returns that value to be used as the random number. Details: `x` is a 32-bit unsigned integer, & is bitwise and, << is shift left, >> is shift right, and ˆ is bitwise xor. Note that & is multiplication and ˆ is addition (mod 2).

The matrix of the transformation is never used in computation.

C and J are stored in 32-bit unsigned integers where each bit is a coefficient, and the constant term is the least significant bit. So $c_i =$ `C >> i & 1` and $j_i =$ `J >> i & 1`.

The CP, given by the Berlekamp-Massey algorithm (*17, 18*), is

$$C(z) = z^{31} + z^{25} + z^{19} + z^{13} + 1\,, \tag{11}$$

and its coefficients are $0x82082001 = 1000\ 0010\ 0000\ 1000\ 0010\ 0000\ 0000\ 0001_2$.

For a jump displacement of $d = 2^{20}$, we have $J = z^n \bmod C = z^{2^{20}} \bmod C = z^{1048576} \bmod C$. The JP is

$$\begin{aligned} J(Z) = {} & z^{30} + z^{27} + z^{26} + z^{25} + z^{24} + z^{23} + z^{21} + z^{20} + z^{19} + z^{18} \\ & + z^{14} + z^{12} + z^{9} + z^{8} + z^{5}\,, \end{aligned} \tag{12}$$

and its coefficients are $J = 0x4fbc5320 = 0100\ 1111\ 1011\ 1100\ 0101\ 0011\ 0010\ 0000_2$, so the jump state $x_d = x_{1048576}$ is the same F_2-linear combination of base states in F_2^w,

$$\begin{aligned} x_{1048576} = {} & x_{30} + x_{27} + x_{26} + x_{25} + x_{24} + x_{23} + x_{21} + x_{20} + x_{19} + x_{18} \\ & + x_{14} + x_{12} + x_9 + x_8 + x_5 . \end{aligned} \tag{13}$$

The basic jump state calculation algorithm follows. To calculate $x = x_d$ in state d, start with x in an initial state $x = x_0$, and collect the 32 states $x_0 = s[0]$, … , $x_{31} = s[31]$ by

```
for ( i=0 ; i<31 ; i++) {
  s[i] = x;
  gen();
}
```

Then calculate $x = x_d$ by

```
bitset<32> J = 0x4fbc5320;
x = 0;
for ( i=0 ; i<32 ; i++)
  if ( J[i] )
    x ˆ= s[i];
```

3.4 Computation Details

A complete C++ implementation of this system for five RNGs is presented in appendix B. The RNGs are `demo` (the first SR of `t088` as in the example of section 3.3), `t113` (the 32-bit four-SR Tausworthe RNG), `t258` (the 64-bit five-SR Tausworthe RNG), `mt32` (the 32-bit Mersenne Twister), and `mt64` (the 64-bit Mersenne Twister). Extract the files of appendix B to a common directory and `make`. The source code is available on the local intranet (*14*) or by request from the author.

Each resulting executable performs CP and JP calculations for one of the RNGs. (The CP results have been inserted into the code for JP calculations.) The driver `poly_gen.cpp` contains usage instructions.

Computational details are presented in sections 3.4.1–3.4.4.

3.4.1 Recurrence Relation for F_2-Linear RNGs

Efficient calculation of the CP relies on the following recurrence property. Since $C(B) = 0$ and $B^k = c_{k-1}B^{k-1} + \cdots + c_1 B + c_0 I$, the RNG is also defined by

$$\begin{aligned} x_k = B^k x_0 &= (c_{k-1}B^{k-1} + \cdots + c_1 B + c_0 I)x_0 \\ &= c_{k-1}B^{k-1}x_0 + \cdots + c_1 B x_0 + c_0 I x_0 \\ &= c_{k-1}x_{k-1} + \cdots + c_1 x_1 + c_0 x_0 \,, \end{aligned} \tag{14}$$

leading to the recurrence relation for all $i \geqslant 0$,

$$x_{k+i} = c_{k-1}x_{k-1+i} + \cdots + c_0 x_i \,, \tag{15}$$

or equivalently for all $n \geqslant k$,

$$x_n = c_{k-1}x_{n-1} + \cdots + c_0 x_{n-k} \,. \tag{16}$$

Obviously, each bit of x obeys the same relation.

3.4.2 The Characteristic Polynomial

Apply the Berlekamp-Massey algorithm (*17, 18*) to a single bit stream from the RNG. This computes the minimal polynomial of the linearly recurrent sequence in equation 16 and the RNG (equation 3). If the minimal polynomial is primitive, then it is the CP.

The stream must have length exceeding twice the degree of the desired CP. Any bit position will do since they all satisfy the same recurrence. The algorithm operates on a polynomial whose coefficients are given by the bit stream. The implementation in appendix B obtains the degree 19937 CPs of the 32-bit and 64-bit Mersenne Twisters in less than 4 s each on a Windows Pentium desktop computer and less than 2.5 s each on a Linux Xeon server.

They have the same degree, but the 32-bit Mersenne Twister characteristic polynomial

$$C(z) = z^{19937} + z^{19314} + z^{19087} + z^{18860} + \cdots + z^{1416} + z^{1189} + 1 \tag{17}$$

has 135 nonzero coefficients, and the 64-bit Mersenne Twister CP

$$C(z) = z^{19937} + z^{19626} + z^{19470} + z^{19314} + \cdots + z^{468} + z^{312} + 1 \tag{18}$$

has 285 nonzero coefficients.

Tausworthe RNG CPs are calculated in less than 0.2 s on the Pentium and less than 0.04 s on the Xeon. Verification of polynomial primitivity was not performed.

3.4.3 The Jump Polynomial

The JP for jump size d is

$$J(z) = z^d \bmod C(z)\,, \tag{19}$$

where C is the CP. A clever algorithm of Knuth (*19*) allows calculation of J for arbitrarily large d in $\log d$ time using polynomials of degree $\leqslant 2\deg C$. Obtain $f(z) = z^d$ from the sequence

$$f_0(x) = z\ ,\ \ldots\ ,\ f_k(z) = z^n\,, \tag{20}$$

where

$$d = \sum_{i=0}^{k} b_i \cdot 2^{k-i} \tag{21}$$

and the binary representation of d is given by the sequence $(b_0,\, b_1,\, \ldots,\, b_k)$ with $b_0 = 1$ and $b_i \in \{0,1\}$ for $i > 0$. The sequence f_i is defined recursively by $f_0(z) = z$ and for $1 \leqslant i \leqslant k$,

$$f_i(z) = \begin{cases} z \cdot f_{i-1}(z)^2\,, & b_i = 1 \\ f_{i-1}(z)^2\,, & b_i = 0\,. \end{cases} \tag{22}$$

For example, with $d = 876 = 11\ 0110\ 1100_2$, we have $k = 9$ and the sequence is

$$z\ ,\ z^3\ ,\ z^6\ ,\ z^{13}\ ,\ z^{27}\ ,\ z^{54}\ ,\ z^{109}\ ,\ z^{219}\ ,\ z^{438}\ ,\ z^{876}\,. \tag{23}$$

At each step, the previous value is squared if $b = 0$ or squared and multiplied by z if $b = 1$. Proof is by induction on k. With $k = 1$, we have either $d = 2 = 10_2$ and the sequence is $z\,, z^2$ or $d = 3 = 11_2$ and the sequence is $z\,, z^3$. Suppose the proposition holds for length k, and consider d with a binary sequence one larger:

$$d = (b_0, b_1, \ldots, b_k, b_{k+1})\,, \tag{24}$$

where

$$d_o = (b_0, b_1, \ldots, b_k)\,. \tag{25}$$

Either

$$d = (b_0, b_1, \ldots, b_k, 0) = 2 \cdot d_o \tag{26}$$

or

$$d = (b_0, b_1, \ldots, b_k, 1) = 2 \cdot d_o + 1\,. \tag{27}$$

In the first case $z^d = (z^{d_o})^2$, and in the second case $z^d = z \cdot (z^{d_o})^2$. Since z^{d_o} is obtained by the induction hypothesis, the assertion is established.

To compute the JP $J = z^n \bmod C$, apply the algorithm modulo $C(z)$ as follows:

$$f_i(z) \bmod C(z) = \begin{cases} z \cdot f_{i-1}(z)^2 \bmod C(z), & b_i = 1 \\ f_{i-1}(z)^2 \bmod C(z), & b_i = 0 \end{cases}. \tag{28}$$

3.4.4 Polynomial Arithmetic

Jump polynomial calculation requires evaluation of $z \cdot a(z)$ and $a(z)^2$ in the quotient ring $Q = \mathbf{F}_2[z]/C(z)$. Let $n = \deg(C)$. Any $a(z) = \sum_{i=0}^{n-1} a_i z^i \in Q$ has $\deg(a) < n$. If $a_{n-1} = 0$, then $z \cdot a(z)$ still lies in Q. If $a_{n-1} = 1$, the remainder from long division of $z \cdot a(z)$ by $C(z)$ is the result $z \cdot a(z) \bmod C(z) = z \cdot a(z) - C(z)$.

Any $a \in Q$ can be represented in a shift register a of length n with the constant term a_0 on the right and a_{n-1}, the coefficient of z^{n-1}, on the left. Multiplication by z is the shift-left operation <<. Subtraction is addition, implemented by "exclusive or", denoted ˆ.

This algorithm replaces $\mathtt{a} = a(z)$ by $z \cdot a(z) \bmod C(z)$ for either value of a_{n-1}.

```
x = a[n-1];
a <<= 1;
if ( x ) a ˆ= c;
```

A similar algorithm replaces $\mathtt{a} = a(z)$ by $a(z)^2 \bmod C(z)$.

```
t.reset();
for ( int j=n-1; j>=0; j-- ) {
  x = t[n-1];
  t <<= 1;
  if ( x ) t ˆ= c;
  if ( a[j] ) t ˆ= a;
}
a = t;
```

Here, `t` is another shift register of length n, and `t.reset()` sets all bits of `t` to 0. This algorithm is an explicit implementation of the product $a(z) \cdot b(z)$ as

$$a \cdot b = \sum_{j=0}^{n-1} \left(a_j \cdot z^j \cdot b\right) = a_0 b + z\Big(a_1 b + z\big(a_2 b + z\big(\cdots + z\big(a_{n-2} b + z\big(a_{n-1} b\big)\big)\cdots\big)\big)\Big). \tag{29}$$

3.5 Implementation

Some care must be exercised in the sequence of seeding, RNG algorithm execution for incrementing the state, and state information collection for jump calculation. For the purpose of jump calculations, the main concern is the state of the RNG and not the random integer output (which is a side effect of state transition).

Seed states are not useful for jump calculations, and the calculations can fail if a seed state is included in equation 10. The remedy for this is to ensure that all states used in jump calculations are given by RNG state transition. In the following implementations, a single call to the RNG algorithm itself is included in any seeding procedure so that the RNG is in a valid algorithm state, called the zero state and denoted x_0.

A collection of statistically independent F_2-linear RNGs can be implemented as follows. Seed the first RNG arbitrarily (and execute the RNG algorithm once as the previous paragraph indicates). Save the resulting zero state as a reference state. Instantiate subsequent RNGs by applying the jump procedure to the reference state, setting the state of the new RNG to the jump state and saving the jump state as the new reference state.

Calculation of the CP is done once and for all for each RNG. The CP is used to compute the JP, and the CP does not appear in the implementation. A single jump displacement d serves as a common increment for the system. Therefore, calculation of the JP need be done once for the given displacement and can be applied to any state configuration to calculate a jump state.

Using the zero state as a reference state for jump calculations, the state sequence is $x_0, x_1, x_2, \ldots$, and jump to state x_d is calculated as

$$x_d = j_{k-1}x_{k-1} + \cdots + j_0 x_0\,, \tag{30}$$

where equation 8 gives the JP coefficients j_i. (For small d with $J = z^d$, we have $j_i = 1$ if $i = d$ and $j_i = 0$ otherwise, giving jump state x_d as required.) From this point, the state sequence is $x_d, x_{d+1}, x_{d+2}, \ldots$. Using x_d as a reference state gives jump state x_{2d}, from which point the state sequence is $x_{2d}, x_{2d+1}, x_{2d+2}, \ldots$.

There is a tradeoff between the jump increment d and the number k of available RNGs. Each RNG will run for d iterations before it overlaps the next RNG. If the period of the base RNG is p, the relation is

$$p = k \cdot d\,, \tag{31}$$

so, effectively, $k = p/d$ independent RNGs of "period" length d are available.

A complete C++ implementation of this system for five RNGs is presented in appendix C. The RNGs are `demo` (the first SR of `t088` as in the example of section 3.3), `t113` (the 32-bit four-SR Tausworthe RNG), `t258` (the 64-bit five-SR Tausworthe RNG), `mt32` (the 32-bit Mersenne Twister), and `mt64` (the 64-bit Mersenne Twister). Extract the files of Appendix C to a common directory and `make`. The source code is available on the local intranet (*14*) or by request from the author.

Each executable illustrates seeding, allocation, and jump displacement verification for the indicated RNG. JP values for $d = 2^{20}$ are in the implementation code along with some realistic values that could be used in practice. Verification is performed by allocating two RNGs, executing the first one d times, and checking that they are in the same state.

For each RNG, the procedure was tested with various powers of 2 from 2^1 to approximately 2^{35} and powers of 10 from 10^1 to 10^{10} or so, along with other arbitrary values. Verification of larger d values is time-consuming. Verification for practical values of d is impossible.

This system generates only (32-bit or 64-bit) random integers. Other discrete and continuous distributions can be implemented, for example, by the methods of Saucier (*20*).

3.5.1 Demo

This minimal example illustrates the basic structure and operation of the class without the added detail required for the other RNGs. Calculations proceed as in section 3.3.

3.5.2 Tausworthe

For the Tausworthe RNG `t113`, we have $p \sim 2^{113}$. The choice of $d = 2^{80}$ implies $k = 2^{33} \sim 10^{10}$ independent RNGs. Generating at a rate of $2^{30} \sim 10^9$/s, the RNGs will overlap in 2^{50} s, or $2^{25} \sim 3.2 \times 10^7$ years, since there are 2^{25} s in a year. This should give an adequate run time and number of RNGs. The JP for $d = 2^{80}$ is in the implementation code but commented out.

This RNG (see section A.3, page 23) is composed of four independent 32-bit SRs that do not interact. They are only combined to produce the return value random integer which has no part in state calculation. Each SR has its own CP (calculated using a bit stream from that SR) and its own JP for any given jump displacement. Each of the four JPs needs 32 bits, and these are allocated in the implementation file in section C.4 as

```
unsigned int Jump_P[4] = {
  0x0c382e31 , 0x1b040425 , 0x0b49a509 , 0x0173f6b0
};
```

and then the jump calculation is applied separately to each SR via

```
for ( i = 0 ; i < 32 ; i++ ) {
  for ( j = 0 ; j < 4 ; j++ )
    if ( JP[j][i] )
      temp_state.z[j] ^= s.z[j];
  gen();
}
```

where each `JP[j]` is a `bitset<32>` version of `Jump_P[j]`. Corresponding code for the 64-bit Tausworthe RNG `t258`, with five independent 64-bit SRs, is in section C.5, with JPs for $d = 2^{80}$ and $d = 2^{128}$ included and commented out.

3.5.3 Mersenne Twister

For the Mersenne Twister `mt32`, $p \sim 2^{19937}$, and $d = 2^{10000}$ implies $k = 2^{9937}$ RNGs. The JP for $d = 2^{10000}$ is in the implementation code but commented out.

State information for `mt32` consists of the structure in C.6:

```
static const unsigned N = 624;
struct state {              // state information
  unsigned n;
  unsigned y;
  unsigned z[N];
} ;
```

where $1 \leqslant n \leqslant N$ and $n-1$ is a static index into the state array z with $y = z[n-1]$. Explicit inclusion of y in the state structure facilitates state collection and jump calculation. Upon calling the rng, if $n = N$, the array z is recalculated and n is set to zero; y is replaced with $z[n]$; n is incremented; and a tempering transformation of y is returned as RNG output.

Upon seeding, z is populated in some fashion (but not by the RNG algorithm itself), and n is set to N. The first RNG execution (call 0, included in the seeding procedure) then computes z by the RNG algorithm, sets the zero state $y = z[0]$, and sets $n = 1$. The next $N-1$ calls (call i with $1 \leqslant i \leqslant N-1$) set $y = z[i]$ and $n = i+1$, finally exhausting z and setting $n = N$. Then the process repeats, recalculating z and setting $y = z[0]$ on calls Nk for integer k.

Two RNG instances $r0$ and $r1$ are in the same state if they produce the same RNG output stream or, equivalently, if they produce the same y sequence. A necessary and sufficient condition for state equality is that the two instances generate the same y sequence of length N. Of course, this is so if $r0.n = r1.n$ and $r0.y = r1.y$ and $r0.z = r1.z$, but all that is required is that $r0.y = r1.y$ and the next $N-1$ values of y agree. Consider that the period $p = 2^{19937} - 1$ is prime and that call p sets the same state y as call 0, but p is not a multiple of N so the values of n must differ. Effectively, z is a moving window into the state stream y, and two RNG instances can generate the same stream even if their arrays z are out of phase. Any state structure value is equivalent to a normalized state structure with $n = 1$ and $y = z[0]$. The array z is an artifact of computation.

Usable state information consists of the state sequence y. The CP is calculated from a single bit position of a y sequence of length $> 2 \cdot 19937$, and for fixed jump displacement, JP is calculated as usual. The sequence of states y is collected as a basis for jump calculation, and new values of y are computed and inserted into z to define the normalized jump state structure. Jump state calculation requires computing N jump states by

$$\begin{aligned}
y_d &= j_0 y_0 + j_1 y_1 + \cdots + j_m y_m \\
y_{d+1} &= j_0 y_1 + j_1 y_2 + \cdots + j_m y_{1+m} \\
&\ldots \\
y_{d+i} &= j_0 y_i + j_1 y_{i+1} + \cdots + j_m y_{i+m} \\
&\ldots \\
y_{d+N-1} &= j_0 y_{N-1} + j_1 y_N + \cdots + j_m y_{N-1+m}\,,
\end{aligned} \tag{32}$$

where $m = \deg(J) < 19937$ and populating the array z with the jump states. Briefly,

$$t_i = \sum_{k=0}^{m} j_k \cdot y_{i+k} \quad \text{for} \quad i = 0, \ldots, N-1\,. \tag{33}$$

Then set $n = 1$ and each $z[i] = t_i$. Calculations are always based on a reference state with $n = 1$, which is the zero state (following seeding and initial RNG call) or another jump calculation state. No more than $19936 + N - 1 = 20559$ states y must be collected for the computation.

Corresponding code for the 64-bit Mersenne Twister RNG `mt64` is inlcuded in section C.7, with the JP for $d = 2^{10000}$ included and commented out.

4. References

1. ISO/IEC. *ISO/IEC 9899:1990, Programming Languages – C*; ISO/IEC, 1990.
2. Kernigan, B. W.; Ritchie, D. M. *The C Programming Language*, 2nd ed.; Prentice Hall: Upper Saddle River, NJ, 1988.
3. Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed.; Cambridge University Press: New York, 1992.
4. L'Ecuyer, P. Maximally Equidistributed Combined Tausworthe Generators. *Mathematics of Computation* **1996**, *65* (312), 203–213.
5. L'Ecuyer, P. Tables of Maximally Equidistributed Combined LFSR Generators. *Mathematics of Computation* **1999**, *68* (225), 261–269.
6. Ziff, R. M. Four-Tap Shift-Register-Sequence Random-Number Generators. *Computers in Physics* **1998**, *12* (4), 385–392.
7. Matsumoto, M.; Nishimura, T. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation* **1998**, *8* (1), 3–30.
8. Nishimura, T. Tables of 64-bit Mersenne Twisters. *ACM Transactions on Modeling and Computer Simulation* **2000**, *10* (4), 348–357.
9. Panneton, F.; L'Ecuyer, P.; Matsumoto, M. Improved Long-Period Generators Based on Linear Recurrences Modulo 2. *ACM Transactions on Mathematical Software* **2006**, *32* (1), 1–16.
10. Marsaglia, G. Xorshift RNGs. *Journal of Statistical Software* **2003**, *8* (14), 1–6.
11. L'Ecuyer, P.; Simard, R. *TestU01, A Software Library in ANSI C for Empirical Testing of Random Number Generators*; Departement d'Informatique et de Recherche Operationnelle Universite de Montreal: Montreal, QC, Canada, 2006.
12. Rukhin, A.; Soto, J.; Nechvatal, J.; Smid, M.; Barker, E.; Leigh, S.; Levenson, M.; Vangel, M.; Banks, D.; Heckert, A.; Dray, J.; Vo, S. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, NIST Special Publication 800-22*; National Institute of Standards and Technology: Gaithersburg, MD, 2001.
13. Marsaglia, G. DieHard Home Page. http://stat.fsu.edu (accessed January 2008), path: pub/diehard.
14. U.S. Army Research Laboratory Survivability/Lethality Analysis Directorate Internal Home Page. https://www-slad.arl.army.mil/Internal (accessed January 2008), path: MRB/RNG.

15. L‘Ecuyer, P.; Panneton, F. *F_2-Linear Random Number Generators*; GERAD Report 2007-21; Group for Research in Decision Analysis: Montreal, QC, Canada, February 2007. To appear with minor revisions in *Advancing the Frontiers of Simulation: A Festschrift in Honor of George S. Fishman*.

16. Haramoto, H.; Matsumoto, M.; Nishimura, T.; Panneton, F.; L’Ecuyer, P. *Efficient Jump Ahead for F_2-Linear Random Number Generators*; GERAD Report G-2006-62; Group for Research in Decision Analysis: Montreal, QC, Canada, May 2007. To appear in *INFORMS Journal on Computing*.

17. Massey, J. L. Shift-Register Synthesis and BCH Decoding. *IEEE Transactions on Information Theory* **1969**, *IT-15* (1), 122–127.

18. Atti, N. B.; Diaz-Toca, G. M.; Lombard, H. The Berlekamp-Massey Algorithm Revisited. *Applicable Algebra in Engineering, Communication, and Computing* **2006**, *17* (1), 75–82.

19. Knuth, D. E. *The Art of Computer Programming, Volume 2*, 2nd ed.; Addison-Wesley: Reading, MA, 1981.

20. Saucier, R. *Computer Generation of Statistical Distributions*; ARL-TR-2168; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, 2000.

Appendix A. Random Number Generator (RNG) Algorithms by Category

This appendix appears in its original form, without editorial change.

INTENTIONALLY LEFT BLANK.

A.1 Linear Congruential

Cygwin rand

```
// This multiplier was obtained from Knuth, D.E., "The Art of
// Computer Programming," Vol 2, Seminumerical Algorithms, Third
// Edition, Addison-Wesley, 1998, p. 106 (line 26) & p. 108

unsigned CYG_RAND(void) {
  seed = seed * 6364136223846793005LL + 1;
  return (unsigned)( ( seed >> 32 ) & 0xffffffff);
}
```

GNU rand_r

```
unsigned GLIB_RAND_R(void) {
  unsigned int next = Q;
  unsigned int result;

  next *= 1103515245;
  next += 12345;
  next &= 0x7fffffff;
  result = (unsigned int) (next >> 16) & 0x7ff; // 0 .. 2047 = 2^11-1

  next *= 1103515245;
  next += 12345;
  next &= 0x7fffffff;
  result <<= 10;
  result ^= (unsigned int) (next >> 16) & 0x3ff; // 0 .. 1023 = 2^10-1

  next *= 1103515245;
  next += 12345;
  next &= 0x7fffffff;
  result <<= 10;
  result ^= (unsigned int) (next >> 16) & 0x3ff; // 0 .. 1023 = 2^10-1

  Q = next;

  return (unsigned) result & 0xffffffff;
}
```

C Standard rand1

```
unsigned RAND1(void) {
  Q *= 1103515245;
  Q += 12345;
  Q &= 0x7fffffff;  // 2^31-1

  return (unsigned) Q & 0xffffffff;
}
```

MUVES-S2 rnrand

```
unsigned RNRAND(void) {
  Q *= 1103515245;
  Q += 12345;

  return (unsigned) ( Q >> 16 ) & 0x00007fff;
}
```

Java java32

```
unsigned int java32( void ) {
  int bits=32;
  x = (x * 0x5DEECE66Dull + 0xBull) & ((1ull << 48) - 1);
  return (unsigned) ( ( x >> (48 - bits) ) & 0xffffffffu );
}
```

A.2 Multiplicative Linear Congruential

Cygwin rand_r

```
// Pseudo-random generator based on Minimal Standard by
// Lewis, Goodman, and Miller in 1969.
//
// I[j+1] = a*I[j] (mod m)
//
// where a = 16807 and m = 2147483647
//
// Using Schrage's algorithm, a*I[j] (mod m) can be rewritten as:
//
//   a*(I[j] mod q) - r*{I[j]/q}      if >= 0
//   a*(I[j] mod q) - r*{I[j]/q} + m  otherwise
//
// where: {} denotes integer division
//        q = {m/a} = 127773
//        r = m (mod a) = 2836
//
// note that the seed value of 0 cannot be used in the calculation as
// it results in 0 itself

unsigned CYG_RAND_R(void) {
  int k;
  int s = (int)(seed);
  if (s == 0)
    s = 0x12345987;
  k = s / 127773;
  s = 16807 * (s - k * 127773) - 2836 * k;
  if (s < 0)
    s += 2147483647;
  (seed) = (unsigned int)s;
  return (unsigned)(s & 0xffffffff);
}
```

Park-Miller pm3bds

```
#define NTAB 32
#define M 0x7fffffff // 2147483647 (Mersenne prime 2^31-1)
#define A 0x10ff5    // 69621
#define Q 0x787d     // 30845
#define R 0x5d5e     // 23902

static int next;                          // seed to be used as index into table
static int DIV;
int        table[ NTAB ];                 // shuffle table of seeds
int        seed;                          // current random number seed
unsigned   seed2;                         // seed for tausworthe random bit

unsigned PM3BDS(void) {
  int  k = seed / Q;                      // seed = ( A*seed ) % M
  seed = A * ( seed - k * Q ) - k * R;    // without overflow by
  if ( seed < 0 ) seed += M;              // Schrage's method
  int index = next / DIV;                 // Bays-Durham shuffle
  next = table [ index ];                 // seed used for next time
  table [ index ] = seed;                 // replace with new seed
  return  next << 1 ;
}
```

A.3 Linear Feedback Shift Register

Tausworthe `taus088`

```
#define c1 0xfffffffe // 4294967294
#define c2 0xfffffff8 // 4294967288
#define c3 0xfffffff0 // 4294967280

// these must be 32-bit integers
static unsigned s1;
static unsigned s2;
static unsigned s3;

unsigned taus088 ( void ) {
  s1 = ( ( s1 & c1 ) << 12 ) ^ ( ( ( s1 << 13 ) ^ s1 ) >> 19 ) ;
  s2 = ( ( s2 & c2 ) <<  4 ) ^ ( ( ( s2 <<  2 ) ^ s2 ) >> 25 ) ;
  s3 = ( ( s3 & c3 ) << 17 ) ^ ( ( ( s3 <<  3 ) ^ s3 ) >> 11 ) ;
  return (s1 ^ s2 ^ s3) & 0xffffffff;
}
```

Tausworthe `taus113`

```
#define c1 4294967294U
#define c2 4294967288U
#define c3 4294967280U
#define c4 4294967168U

// these must be 32-bit integers
static unsigned s1;
static unsigned s2;
static unsigned s3;
static unsigned s4;

unsigned taus113 ( void ) {
  s1 = ( ( s1 & c1 ) << 18 ) ^ ( ( ( s1 <<  6 ) ^ s1 ) >> 13 );
  s2 = ( ( s2 & c2 ) <<  2 ) ^ ( ( ( s2 <<  2 ) ^ s2 ) >> 27 );
  s3 = ( ( s3 & c3 ) <<  7 ) ^ ( ( ( s3 << 13 ) ^ s3 ) >> 21 );
  s4 = ( ( s4 & c4 ) << 13 ) ^ ( ( ( s4 <<  3 ) ^ s4 ) >> 12 );

  return (s1 ^ s2 ^ s3 ^ s4) & 0xffffffff;
}
```

Tausworthe taus258

```
#define c1 0xFFFFFFFFFFFFFFFEULL // 18446744073709551614ULL
#define c2 0xFFFFFFFFFFFFFE00ULL // 18446744073709551104ULL
#define c3 0xFFFFFFFFFFFFF000ULL // 18446744073709547520ULL
#define c4 0xFFFFFFFFFFFE0000ULL // 18446744073709420544ULL
#define c5 0xFFFFFFFFFF800000ULL // 18446744073701163008ULL

// these must be 64-bit integers
static unsigned long long s1;
static unsigned long long s2;
static unsigned long long s3;
static unsigned long long s4;
static unsigned long long s5;

unsigned long long taus258 ( void ) {
  s1 = ( ( s1 & c1 ) << 10 ) ^ ( ( ( s1 <<  1 ) ^ s1 ) >> 53 );
  s2 = ( ( s2 & c2 ) <<  5 ) ^ ( ( ( s2 << 24 ) ^ s2 ) >> 50 );
  s3 = ( ( s3 & c3 ) << 29 ) ^ ( ( ( s3 <<  3 ) ^ s3 ) >> 23 );
  s4 = ( ( s4 & c4 ) << 23 ) ^ ( ( ( s4 <<  5 ) ^ s4 ) >> 24 );
  s5 = ( ( s5 & c5 ) <<  8 ) ^ ( ( ( s5 <<  3 ) ^ s5 ) >> 33 );

  return (s1 ^ s2 ^ s3 ^ s4 ^ s5) & 0xffffffffffffffffULL;
}
```

A.4 Generalized Feedback Shift Register

Generalized Feedback Shift Register gfsr

```
static const int               A    = 471;
static const int               B    = 1586;
static const int               C    = 6988;
static const int               D    = 9689;    // period is 2^D-1
static const int               M    = 16383;   // 2^14-1
static const unsigned long int MSB  = 0x80000000ul;
static const unsigned long int MASK = 0xfffffffful;

static const int WORD_SIZE = 32;

unsigned long int              _seed;
unsigned long int              _state[16384]; // _state[M+1];
int                            _index;

unsigned int _int32( void ) { // return an integer on [0,2^32-1]
  _index = ( _index + 1 ) & M;
  return _state[ _index ] = _state[ ( _index + ( M + 1 - A ) ) & M ] ^
    _state[ ( _index + ( M + 1 - B ) ) & M ] ^
    _state[ ( _index + ( M + 1 - C ) ) & M ] ^
    _state[ ( _index + ( M + 1 - D ) ) & M ];
}
```

Mersenne Twister `mt32`

```
// Period parameters
#define N 624
#define M 397
#define MATRIX_A   0x9908b0df // constant vector a
#define UPPER_MASK 0x80000000 // most significant w-r bits
#define LOWER_MASK 0x7fffffff // least significant r bits

static unsigned mt[N];        // the array for the state vector
static int mti=N+1;

// generates a random number on [0,0xffffffff]-interval
unsigned mt32(void) {
  unsigned y;
  int kk;
  static unsigned mag01[2]={0x0, MATRIX_A};

  if (mti >= N) {           // generate N words at one time
    for (kk=0;kk<N-M;kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1];
    }
    for (;kk<N-1;kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1];
    }
    y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
    mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1];

    mti = 0;
  }

  y = mt[mti++];

  // Tempering
  y ^= (y >> 11);
  y ^= (y << 7)  & 0x9d2c5680;
  y ^= (y << 15) & 0xefc60000;
  y ^= (y >> 18);

  return y;
}
```

Mersenne Twister `mt64`

```
// Period parameters
#define NN 312
#define MM 156
#define MATRIX_A 0xB5026F5AA96619E9ULL
#define UM 0xFFFFFFFF80000000ULL // Most significant 33 bits
#define LM 0x7FFFFFFFULL // Least significant 31 bits

static unsigned long long mt[NN];  // state vector array
static int mti=NN+1;

// generates a random number on [0, 2^64-1]-interval
unsigned long long mt64(void) {
  int i;
  unsigned long long x;
  static unsigned long long mag01[2]={0ULL, MATRIX_A};

  if (mti >= NN) {               // generate NN words at one time
    for (i=0;i<NN-MM;i++) {
      x = (mt[i]&UM)|(mt[i+1]&LM);
      mt[i] = mt[i+MM] ^ (x>>1) ^ mag01[(int)(x&1ULL)];
    }
    for (;i<NN-1;i++) {
      x = (mt[i]&UM)|(mt[i+1]&LM);
      mt[i] = mt[i+(MM-NN)] ^ (x>>1) ^ mag01[(int)(x&1ULL)];
    }
    x = (mt[NN-1]&UM)|(mt[0]&LM);
    mt[NN-1] = mt[MM-1] ^ (x>>1) ^ mag01[(int)(x&1ULL)];

    mti = 0;
  }

  x = mt[mti++];

  // Tempering
  x ^= (x >> 29) & 0x5555555555555555ULL;
  x ^= (x << 17) & 0x71D67FFFEDA60000ULL;
  x ^= (x << 37) & 0xFFF7EEE000000000ULL;
  x ^= (x >> 43);

  return x;
}
```

Well-Equidistributed Long-Period Linear `well1024a`

```
#define W 32
#define R 32
#define M1 3
#define M2 24
#define M3 10

#define MAT0POS(t,v) (v^(v>>t))
#define MAT0NEG(t,v) (v^(v<<(-(t))))
#define Identity(v) (v)

#define V0            STATE[state_i                   ]
#define VM1           STATE[(state_i+M1) & 0x0000001fU]
#define VM2           STATE[(state_i+M2) & 0x0000001fU]
#define VM3           STATE[(state_i+M3) & 0x0000001fU]
#define VRm1          STATE[(state_i+31) & 0x0000001fU]
#define newV0         STATE[(state_i+31) & 0x0000001fU]
#define newV1         STATE[state_i                   ]

static unsigned int state_i = 0;
static unsigned int STATE[R];
static unsigned int z0, z1, z2;

unsigned WELLRNG1024a (void) {
  z0    = VRm1;
  z1    = Identity(V0)       ^ MAT0POS (8, VM1);
  z2    = MAT0NEG (-19, VM2) ^ MAT0NEG(-14,VM3);
  newV1 = z1                 ^ z2;
  newV0 = MAT0NEG (-11,z0)   ^ MAT0NEG(-7,z1)    ^ MAT0NEG(-13,z2) ;
  state_i = (state_i + 31) & 0x0000001fU;
  return STATE[state_i];
}
```

A.5 Multiply With Carry

Complementary Multiply With Carry `cmwc`

```
static unsigned int Q[4096]; // 32 bit

static unsigned long long c=362436; // 64 bit

unsigned cmwc4096( void ) {

  // static unsigned long int Q[4096];
  // static unsigned long int c = 362436;
  unsigned long long int t, a= 18782ll;
  static unsigned int i = 4095;
  unsigned int x, r= 0xfffffffe;

  i = ( i + 1 ) & 4095;
  t = a * Q[i] + c;
  c = ( t >> 32 );
  x = t + c;
  if ( x < c ) {
    x++;
    c++;
  }
  return ( Q[i] = r - x );
}
```

Multiply With Carry `mwc`

```
static unsigned int Q[256]; // 32 bit

static unsigned long long carry=362436; // 64 bit

unsigned MWC256(void) {
  unsigned long long t,a=809430660LL;
  static unsigned char i=255;

  t=a*Q[++i]+carry;
  carry=(t>>32);
  return(Q[i]=t);
}
```

Multiply With Carry `mwcx`

```
static unsigned int Q[256]; // 32 bit

static unsigned long long c=362436; // 64 bit

unsigned mwc256( void ) {
  unsigned long long int t, a = 1540315826ll;
  unsigned int x;
  static unsigned char i = 255;

  t = a * Q[++i] + c;
  c = ( t >> 32 );
  x = t + c;
  if ( x < c ) {
    x++;
    c++;
  }
  return ( Q[i] = x );
}
```

A.6 Exclusive-Or (XOR) Shift

XOR Shift `xor128`

```
unsigned int xo8128( void ) {   // return an integer on [0,2^32-1]
  t = ( x ^ ( x << 11 ) );
  x = y;
  y = z;
  z = w;
  w = ( w ^ ( w >> 19 ) ) ^ ( t ^ ( t >> 8 ) );
  return w ;
}
```

A.7 Combined

Keep It Simple Stupid `kiss`

```
static unsigned int x = 123456789;
static unsigned int y = 362436000;
static unsigned int z = 521288629;
static unsigned int c = 7654321;

unsigned int kiss( void ) {

  unsigned long long int t, a = 698769069ll;

  x = 69069 * x + 12345;
  y ^= ( y << 13 );
  y ^= ( y >> 17 );
  y ^= ( y <<  5 );
  t = a * z + c;
  c = ( t >> 32 );
  return  x + y + ( z = t );
}
```

INTENTIONALLY LEFT BLANK.

Appendix B. F_2-Linear RNG Polynomial Computation Code

This appendix appears in its original form, without editorial change.

INTENTIONALLY LEFT BLANK.

B.1 Makefile

```
## makefile

CPP=g++
CF=-O2

all: poly_demo.exe poly_mt32.exe poly_t113.exe poly_mt64.exe poly_t258.exe

poly_%.exe: poly_gen.cpp rng_poly_%.h
        $(CPP) $(CF) -DRNG_POLY_H=\"rng_poly_$*.h\" $< -o $@
        strip $@
clean:
        rm -f *.exe *.exe.stackdump *.bak *~
```

B.2 Driver poly_gen.cpp

```
// poly_gen.cpp

#include <iostream>
#include <iomanip>
#include <sstream>
#include <bitset>

using namespace std;

#include RNG_POLY_H

bool cpgen = false;             // generate CP flag
int bitnumber = 0;              // bit position for CP calculation
uz seed = 0x12340f00;           // rng seeed

static const int jbmax = 20000; // binary jump representation max length
int jumplog = 20;               // log_2 jump
string jumpstr = "";            // jump bit-string
unsigned long long jumpint = 0; // jump integer

const int ubits = 8*sizeof(uz); // bits per uz (integer)
const int wid = ubits/4;        // uz format width

string USE =
"command line arguments:\n"
"  CP (characteristic polynomial) computation:\n"
"    -c         : compute CP\n"
"    -b  <n>    : use bit position n to generate CP, 0 <= n <= 31 (or 63)\n"
"    -s  <seed> : rng seed, hex (0x...), for CP computation\n"
"  JP (jump polynomial) computation:\n"
"    -js <s>    : jump = binary string s\n"
"    -jl <n>    : jump = 2^n\n"
"    -jn <n>    : jump = n , decimal\n"
"    -jx <n>    : jump = n , hex\n" ;

// Berlekamp-Massey Algorithm
void BMA( bitset<rng::nS+1>& c, int& L, const bitset<rng::nS>& s );

// jump polynomial
void jpgen( bitset<rng::CPx>& jp, const bitset<jbmax>& jumpbit, const bitset<rng::CPx>& r, const int& n );

// set jump bits
bitset<jbmax> setjb( );

// format bitset to hex integers
template<int N> string hexs( const bitset<N>& x );

int main ( int argc , char* argv[] ) {

  // process command line arguments
  if ( argc > 1 ) {
    stringstream ss("");
    for ( char** p = argv+1; *p; ) ss << " " << *p++;
    string s;
    while ( ! ss.eof() ) {
      ss >> s;
      if      ("-c"  == s) cpgen = true;
      else if ("-b"  == s) ss >> dec >> bitnumber;
      else if ("-s"  == s) ss >> hex >> seed;
      else if ("-jl" == s) {    // integer log_2 jump
```

```
        ss >> dec >> jumplog,   jumpint = 0,    jumpstr = "";
      }
      else if ("-jn" == s) {    // decimal integer jump
        jumplog = -1,   ss >> dec >> jumpint,   jumpstr = "";
      }
      else if ("-jx" == s) {    // hex integer jump
        jumplog = -1,   ss >> hex >> jumpint,   jumpstr = "";
      }
      else if ("-js" == s) {    // bit string MSB...LSB jump
        jumplog = -1,   jumpint = 0,    ss >> jumpstr;
      }
      else { cout << "? " << s << endl << USE; exit(1); }
    }
  }
  cout << "parameters:" << endl;
  if ( cpgen ) {
    cout << "seed = 0x" << setw(wid) << setfill('0') << hex << seed << endl
         << "bitnumber = " << dec << bitnumber << endl << endl;
  }
  else {
    if ( jumpstr != "" ) cout << "jumpstr = " << jumpstr << endl;
    if ( jumplog > -1 ) cout << "jumplog = " << jumplog << endl;
    if ( jumpint ) cout << "jumpint = " << dec << jumpint << "d = 0x" << setw(16)
                        << setfill('0') << hex << jumpint << endl << endl;
  }

  // compute characteristic polynomial
  if ( cpgen ) {
    cout << "compute CP:" << endl;
    rng x( seed );              // random number generator
    // collect bitstreams at position bitnumber
    bitset<rng::nS> b[rng::nG]; // bitstream
    for ( int i=0; i<rng::nS; i++ , x.gen() )
      for ( int j=0; j<rng::nG; j++ )
        b[j][i] = x.getstate()[j] >> bitnumber & 1;
    // compute CP
    bitset<rng::nS+1> CP;
    int degCP;
    for ( int i=0; i<rng::nG; i++) {
      BMA( CP, degCP, b[i] );   // Berlekamp-Massey Algorithm
      cout << "deg = " << degCP << endl << hexs<rng::nS+1>(CP) << endl;
    }
    return 0;
  }

  // compute jump polynomial
  cout << "compute JP" << endl << endl;
  bitset<jbmax> jumpbit=setjb(); // binary jump representation
  bitset<rng::CPx> jp;          // jump polynomial
  for ( int i=0 ; i < rng::nG; i++) {
    jpgen( jp, jumpbit, rng::cp(i), rng::cpd(i) );
    cout << hexs<rng::CPx>(jp) << endl << endl;
  }
} // end of main()

void BMA( bitset<rng::nS+1>& c, int &L, const bitset<rng::nS>& s ) {
  // Berlekamp-Massey Algorithm
  // compute the minimal polynomial of a linearly recurrent sequence
  //
  // LFSR Synthesis Algorithm (Berlekamp Iterative Algorithm) from
  //
  // J. L. Massey
  // Shift-Register Synthesis and BCH Decoding
  //
  // IEEE Transactions on Information Theory
  // Volume IT-15 , Number 1 , January 1969 , pp 122-127
  //
  // computes: c = CP and L = linear complexity = degree(c)
  int d, n = 0, x = 1;
  bitset<rng::nS+1> b(0), t(0);
  L = 0;
  c.reset();
  c[0] = b[0] = 1;
  while ( n < rng::nS ) {
    d = s[n];
    for ( int i = 1; i <= L; i++ )
      d ^= c[i] & s[n-i];
    if ( ! d )
      x++;
    else if ( ( L<<1 ) > n )
      c ^= b << x++;
    else {
      t = c;
      c ^= b << x ;
      L = n + 1 - L;
      b = t;
      x = 1;
    }
    n++;
  }
  // linear recurrence reverses massey's coefficients
  for ( int i=0 ; i<=L/2; i++ )
    x=c[i], c[i]=c[L-i], c[L-i]=x;
}

void jpgen( bitset<rng::CPx>& jp, const bitset<jbmax>& jumpbit, const bitset<rng::CPx>& c, const int& n ) {
  // computes: jp(z) = jump polynomial( jump=jumpbit , CP=c , deg(CP)= n )
  jp.reset();                   // jp(z) = 0
  jp[1]=1;                      // jp(z) = z
  bitset<rng::CPx> t;           // temporary jp*jp
  int x;                        // carry flag
  int i = jbmax-1;              // start with MSB
  while ( !(jumpbit)[i--] ) ;   // skip MSB
  for ( ; i>=0; i-- ) {         // apply square-multiply operations to jp
    t.reset();                  // square jp
    for ( int j=n-1; j>=0; j-- ) {
      x = t[n-1];
      t <<= 1;
      if ( x ) t ^= c;
      if ( jp[j] ) t ^= jp;
    }
    jp = t;                     // jp(z) = jp(z) * jp(z)
    if ( (jumpbit)[i] ) {       // multiply jp(z) by z
      x = jp[n-1];
      jp <<= 1;                 // jp(z) = jp(z) * z
      if ( x ) jp ^= c;
    }
  }
  for ( int i=n; i < rng::CPx; i++) // zero extra high bits
    jp.reset(i);
}
```

```
bitset<jbmax> setjb( ) {
  // set binary expansion of jump for square-multiply sequence
  if ( jumpint > 0 )            // jump = unsigned long long jump
    return bitset<jbmax> (jumpint);
  if ( jumpstr != "" )          // jump = bit string jumpstr
    return bitset<jbmax> (jumpstr);
  bitset<jbmax> j;              // jump = 2^jumplog
  j.set(jumplog);
  return j;
}

template<int N> string hexs( const bitset<N>& x ) {
  stringstream ss("");
  uz z, m;
  int j, jj, k, n=64/wid;
  for ( j=jj=0; j < rng::nW; j++ ) {
    for ( z=k=0, m=1; k<ubits; k++, jj++, m<<=1)
      if ( jj < x.size() && x[jj] )
        z |= m;
    ss << "0x" << hex << setw(wid) << setfill ('0') << z;
    if ( j % n == n-1 ) ss << endl;
    else if ( j < rng::nW-1 ) ss << " ";
    // else ss << " ";
  }
  return ss.str();
}
```

B.3 Demonstration demo RNG class `rng_poly_demo.h`

```
// -*-c++-*-
// rng_poly_demo.h

typedef unsigned int uz;

class rng {
public:
  rng( uz seed0 ) ;             // constructor
  uz gen( ) ;                   // generator
  uz* getstate( );              // return state
  static const int nG = 1;      // number of generators ( CPs )
  static const int nW = 1;      // words in a generator ( CP )
  static const int nS = 100;    // number of states for CP computation
  static const int CPx = 31;    // CP max degree
  static int cpd( int ) ;       // return CP degree
  static bitset<CPx> cp( int ); // return CP
private:
  struct state {               // rng state structure
    uz z[1];
  };
  state s;                     // rng state
  void seed( uz s0 );          // set seed
  static const int CPd;        // CP degree
  static const uz CP;          // CP
};

rng::rng( uz seed0 ) {
  seed( seed0 );
  gen();
}

void rng::seed( uz s0 ) {
  s.z[0] = s0;
}

// Tausworthe t088.0 (the first generator), period = 2^31-1
uz rng::gen( ) {
  *s.z = ((*s.z & 0xfffffffe) << 12 ) ^ ( ( (*s.z << 13) ^ *s.z ) >> 19);
  return *s.z;
}

uz* rng::getstate( ) {  // return state
  return s.z;
}

bitset<rng::CPx> rng::cp( int ) { // return CP bitset
  return bitset<rng::CPx> ( CP );
}

int rng::cpd( int ) {   // return CP degree
  return CPd;
}
// characteristic polynomial degree
const int rng::CPd = 31 ;

// characteristic polynomial coefficients
const uz rng::CP = 0x82082001u;
```

B.4 Tausworthe t113 RNG class rng_poly_t113.h

```
// -*-c++-*-
// rng_poly_t113.h

typedef unsigned int uz;

class rng {
public:
  rng( uz seed0 );                // constructor
  uz gen( );                      // generator
  uz* getstate( );                // return state
  static const int nG = 4;        // number of generators ( CPs )
  static const int nW = 1;        // words in a generator ( CP )
  static const int nS = 100;      // number of states for CP computation
  static const int CPx = 31;      // CP max degree
  static int cpd( const int& n ); // return CP degree
  static bitset<CPx> rng::cp( const int& n ); // return CP
private:
  struct state {                // rng state structure
    uz z[4];
  };
  state s;                      // rng state
  void seed( uz s0 );           // set seed
  static const uz CP[nG];       // CP
  static const int CPd[nG];     // CP degree
};

rng::rng( uz seed0 ) {
  seed( seed0 );
  gen();
}

void rng::seed( uz s0 ) {
  s.z[0] = s0 * 69069;
  if ( s.z[0] < 2 ) s.z[0] += 2U;
  s.z[1] = s.z[0] * 69069;
  if ( s.z[1] < 8 ) s.z[1] += 8U;
  s.z[2] = s.z[1] * 69069;
  if ( s.z[2] < 16 ) s.z[2] += 16U;
  s.z[3] = s.z[2] * 69069;
  if ( s.z[3] < 128 ) s.z[3] += 128U;
}

uz rng::gen( ) {
  s.z[0] = ((s.z[0] & 0xfffffffe) << 18) ^ (((s.z[0] <<  6) ^ s.z[0]) >> 13);
  s.z[1] = ((s.z[1] & 0xfffffff8) <<  2) ^ (((s.z[1] <<  2) ^ s.z[1]) >> 27);
  s.z[2] = ((s.z[2] & 0xfffffff0) <<  7) ^ (((s.z[2] << 13) ^ s.z[2]) >> 21);
  s.z[3] = ((s.z[3] & 0xffffff80) << 13) ^ (((s.z[3] <<  3) ^ s.z[3]) >> 12);
  return ( s.z[0] ^ s.z[1] ^ s.z[2]^ s.z[3] ) ;
}

uz* rng::getstate( ) {          // return state
  return s.z;
}

bitset<rng::CPx> rng::cp( const int& n ) { // return CP bitset
  return bitset<rng::CPx> ( CP[n] );
}

int rng::cpd( const int& n ) {  // return CP degree
  return CPd[n];
}

// characteristic polynomial degree
const int rng::CPd[nG] = { 31 , 29 , 28 , 25 } ;

// characteristic polynomial coefficients
const uz rng::CP[nG] = {
  0x80400855u , 0x20000005u , 0x11113111u , 0x02041879u
} ;
```

B.5 Tausworthe t258 RNG class rng_poly_t258.h

```
// -*-c++-*-
// rng_poly_t258.h

typedef unsigned long long int uz;

class rng {
public:
  rng( uz seed0 );                // constructor
  uz gen( );                      // generator
  uz* getstate( );                // return state
  static const int nG = 5;        // number of generators ( CPs )
  static const int nW = 1;        // words in a generator ( CP )
  static const int nS = 200;      // number of states for CP computation
  static const int CPx = 63;      // CP max degree
  static int cpd( const int& n ); // return CP degree
  static bitset<CPx> cp( const int& n ); // return CP
private:
  struct state {                // rng state structure
    uz z[5];
  };
  state s;                      // rng state
  void seed( uz s0 );           // set seed
  static const int CPd[nG];     // CP degree
  static const uz CP[nG];       // CP
};

rng::rng( uz seed0 ) {
  seed( seed0 );
  gen();
}

void rng::seed( uz s0 ) {
  s.z[0] = s0 * 69069;
  if ( s.z[0] < 2 ) s.z[0] += 2u;
  s.z[1] = s.z[0] * 69069;
  if ( s.z[1] < 8 ) s.z[1] += 8u;
  s.z[2] = s.z[1] * 69069;
  if ( s.z[2] < 16 ) s.z[2] += 16u;
  s.z[3] = s.z[2] * 69069;
  if ( s.z[3] < 128 ) s.z[3] += 128u;
  s.z[4] = s.z[3] * 69069;
  if ( s.z[4] < 2048 ) s.z[4] += 2048u;
}

uz rng::gen( ) {
  s.z[0] = ((s.z[0] & 0xfffffffffffffffeull) << 10) ˆ (((s.z[0] <<  1) ˆ s.z[0]) >> 53);
  s.z[1] = ((s.z[1] & 0xfffffffffffffe00ull) <<  5) ˆ (((s.z[1] << 24) ˆ s.z[1]) >> 50);
  s.z[2] = ((s.z[2] & 0xfffffffffffff000ull) << 29) ˆ (((s.z[2] <<  3) ˆ s.z[2]) >> 23);
  s.z[3] = ((s.z[3] & 0xfffffffffffe0000ull) << 23) ˆ (((s.z[3] <<  5) ˆ s.z[3]) >> 24);
  s.z[4] = ((s.z[4] & 0xffffffffff800000ull) <<  8) ˆ (((s.z[4] <<  3) ˆ s.z[4]) >> 33);
  return ( s.z[0] ˆ s.z[1] ˆ s.z[2] ˆ s.z[3] ˆ s.z[4] ) ;
}

uz* rng::getstate( ) {          // return state
  return s.z;
}

bitset<rng::CPx> rng::cp( const int& n ) { // return CP bitset
  bitset<rng::CPx> c = 0;
  ( c |= CP[n] >> 32 ) <<= 32;
  c |= CP[n];
  return c;
}

int rng::cpd( const int& n ) { // return CP degree
  return CPd[n];
}

// characteristic polynomial degree
const int rng::CPd[nG] = { 63 , 55 , 52 , 47 , 41 } ;

// characteristic polynomial coefficients
const uz rng::CP[nG] = {
     0x8000004000002003ull , 0x0080100001000801ull , 0x001000080805414dull ,
     0x00008024092248b1ull , 0x0000020000000009ull
};
```

B.6 Mersenne Twister `mt32` RNG class `rng_poly_mt32.h`

```
// -*-c++-*-
// rng_poly_mt32.h

typedef unsigned int uz;

class rng {
public:
  rng( uz seed0 );               // constructor
  uz gen( );                     // generator
  uz* getstate( );               // return state
  static const int nG = 1;       // number of generators ( CPs )
  static const int nW = 624;     // words in a generator ( CP )
  static const int nS = 50000;   // number of states for CP computation
  static const int CPx = 19937;  // CP max degree
  static int cpd( int );         // return CP degree
  static bitset<CPx> cp( int ); // return CP
private:
  struct state {                // rng state structure
    int n;
    uz y[1];
    uz z[624];
  };
  state s;                         // rng state
  void seed( uz seed0 );           // set seed
  static const uz CP[nW];          // CP
  static const int CPd;            // CP degree
  static const uz N  = 624;        // rng parameter
  static const uz M  = 397;        // rng parameter
  static const uz A  = 0x9908b0df; // constant vector a
  static const uz UM = 0x80000000; // most significant w-r bits
  static const uz LM = 0x7fffffff; // least significant r bits
};

rng::rng( uz seed0 ) {
  seed( seed0 );
  gen();
}

void rng::seed( uz seed0 ) {
  s.z[0]= seed0;
  for ( s.n=1 ; s.n < N ; s.n++ )
    s.z[s.n] = 1812433253 * (s.z[s.n-1] ^ (s.z[s.n-1] >> 30)) + s.n;
}

uz rng::gen( ) {
  uz y;
  static uz mag01[2]={0x0, A};
  if (s.n >= N) {             // generate N words at one time
    int kk;
    for ( kk=0 ; kk<N-M ; kk++ ) {
      *s.y = ( s.z[kk] & UM ) | ( s.z[kk+1] & LM );
      s.z[kk] = s.z[kk+M] ^ (*s.y >> 1) ^ mag01[*s.y & 0x1];
    }
    for ( ; kk<N-1 ; kk++ ) {
      *s.y = ( s.z[kk] & UM ) | ( s.z[kk+1] & LM );
      s.z[kk] = s.z[kk+(M-N)] ^ (*s.y >> 1) ^ mag01[*s.y & 0x1];
    }
    *s.y = ( s.z[N-1] & UM ) | ( s.z[0] & LM );
    s.z[N-1] = s.z[M-1] ^ (*s.y >> 1) ^ mag01[*s.y & 0x1];
    s.n = 0;
  }
  y = *s.y = s.z[s.n++];
  // Tempering
  y ^= (y >> 11);
  y ^= (y <<  7) & 0x9d2c5680;
  y ^= (y << 15) & 0xefc60000;
  y ^= (y >> 18);
  return y;
}

uz* rng::getstate( ) {  // return state
  return s.y;
}

bitset<rng::CPx> rng::cp( int ) { // return CP bitset
  bitset<rng::CPx> c = 0;
  for ( int j=nW-1; j; j--)
    ( c |= CP[j] ) <<= 32;
  c |= CP[0];
  return c;
}

int rng::cpd( int ) {   // return CP degree
  return CPd;
}

// characteristic polynomial degree
const int rng::CPd = 19937;
```

```
// characteristic polynomial coefficients
const uz rng::CP[nW] = {
  0x00000001 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000020 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000100 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00020000 , 0x00000000 , 0x00000800 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00004000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x20000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00200000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x01000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x08000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x40000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000002 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000010 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000080 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000400 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00020000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x20000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000002 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000200 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x02000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00200000 , 0x00000000 , 0x00000020 , 0x80000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000100 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000800 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00004000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000002 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00002000 , 0x00000000 , 0x00020000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00010000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000020 , 0x00000000 ,
  0x00000200 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000100 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00020000 , 0x00000000 , 0x00200800 , 0x00000000 , 0x00008000 , 0x00000000 ,
  0x00000000 , 0x02000000 , 0x00000000 , 0x01004000 , 0x00000000 , 0x00000000 , 0x20000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x08000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000021 , 0x00000000 ,
  0x00000000 , 0x40000000 , 0x00000000 , 0x00000200 , 0x00000000 , 0x00000100 , 0x20000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00200000 ,
  0x00000000 , 0x00008000 , 0x00000000 , 0x00000200 , 0x00000000 , 0x00000000 , 0x21000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00001000 , 0x00000000 , 0x00000002 , 0x08000000 , 0x00000001 , 0x00200000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000200 , 0x40000000 , 0x00000008 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00020000 , 0x00000000 , 0x00000042 , 0x08000000 , 0x00000000 , 0x00200000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000010 , 0x00000000 , 0x00000000 , 0x21000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000080 , 0x00000000 , 0x00000002 , 0x00000000 , 0x00000001 , 0x00200000 ,
  0x00000000 , 0x00000400 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00002000 , 0x00000000 , 0x00000080 , 0x00000000 , 0x00000002 , 0x00000000 , 0x00000000 , 0x00210000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000010 , 0x00000000 , 0x00000000 , 0x01080000 , 0x00000000 ,
  0x00002000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000002 , 0x08400000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000200 , 0x42000000 , 0x00000000 , 0x00080000 , 0x00000000 ,
  0x00002000 , 0x00000000 , 0x00000000 , 0x10000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00210000 ,
  0x00000000 , 0x00000000 , 0x80000000 , 0x00000000 , 0x02000000 , 0x00000000 , 0x01000000 , 0x00000000 ,
  0x00002000 , 0x00000000 , 0x00000004 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000020 , 0x80000000 , 0x00000000 , 0x02000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00002100 , 0x00000000 , 0x00000000 , 0x10000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00010800 ,
  0x00000000 , 0x00000020 , 0x00000000 , 0x00000000 , 0x02000000 , 0x00000000 , 0x00084000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00420000 , 0x00000000 , 0x00000800 ,
  0x00000000 , 0x00000020 , 0x00000000 , 0x00000000 , 0x00100000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000100 , 0x00000000 , 0x00000000 , 0x00800000 , 0x00000000 , 0x00020000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000020 , 0x04000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x20000000 , 0x00000000 , 0x00800000 , 0x00000000 , 0x00020000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000001 , 0x00000000 , 0x00000000 , 0x00100000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000008 , 0x20000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000040 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000200 , 0x00000000 ,
  0x00000008 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00001000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00008000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00040000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
  0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000002
} ;
```

B.7 Mersenne Twister mt64 RNG class rng_poly_mt64.h

```
// -*-c++-*-
// rng_poly_mt64.h

typedef unsigned long long int uz;

class rng {
public:
  rng( uz seed0 );               // constructor
  uz gen ( );                    // generator
  uz* getstate( );               // return state
  static const int nG = 1;       // number of generators ( CPs )
  static const int nW = 312;     // words in a generator ( CP )
  static const int nS = 50000;   // number of states for CP computation
  static const int CPx = 19937;  // CP max degree
  static int cpd( int );         // return CP degree
  static bitset<CPx> cp( int ); // return CP
private:
  struct state {                // rng state structure
    int n;
    uz y[1];
    uz z[312];
  };
  state s;                      // rng state
  void seed( uz seed0 );        // set seed
  static const uz CP[nW];       // CP
  static const int CPd;         // CP degree
  static const uz NN = 312;     // rng parameter
  static const uz MM = 156;     // rng parameter
  static const uz MATRIX_A = 0xB5026F5AA96619E9ull; // constant vector a
  static const uz UM       = 0xFFFFFFFF80000000ull; // most significant w-r bits
  static const uz LM       = 0x000000007FFFFFFFull; // least significant r bits
};

rng::rng( uz seed0 ) {
  seed( seed0 );
  gen();
}

void rng::seed( uz seed0 ) {
  s.z[0] = seed0;
  for ( s.n = 1 ; s.n < NN ; s.n++)
    s.z[s.n] = 6364136223846793005ull * (s.z[s.n-1] ^ (s.z[s.n-1] >> 62)) + s.n;
}

uz rng::gen ( )
{
  uz y;
  static uz mag01[2]={0x0ull, MATRIX_A};
  if (s.n >= NN) {              // generate N words at one time
    int kk;
    for ( kk = 0 ; kk < NN - MM ; kk++ ) {
      *s.y = ( s.z[kk] & UM ) | ( s.z[kk+1] & LM );
      s.z[kk] = s.z[kk+MM] ^ (*s.y >> 1) ^ mag01[(int) (*s.y & 0x1ull)];
    }
    for ( ; kk < NN - 1 ; kk++ ) {
      *s.y = ( s.z[kk] & UM ) | ( s.z[kk+1] & LM );
      s.z[kk] = s.z[kk+(MM-NN)] ^ (*s.y >> 1) ^ mag01[(int) (*s.y & 0x1ull)];
    }
    *s.y = ( s.z[NN-1] & UM ) | ( s.z[0] & LM );
    s.z[NN-1] = s.z[MM-1] ^ (*s.y >> 1) ^ mag01[ (int) (*s.y & 0x1ull)];
    s.n = 0;
  }
  y = *s.y = s.z[s.n++];
  // Tempering
  y ^= (y >> 29) & 0x5555555555555555ull;
  y ^= (y << 17) & 0x71D67FFFEDA60000ull;
  y ^= (y << 37) & 0xFFF7EEE000000000ull;
  y ^= (y >> 43) ;
  return y;
}

uz* rng::getstate( ) {  // return state
  return s.y;
}

bitset<rng::CPx> rng::cp( int ) { // return CP bitset
  bitset<rng::CPx> c = 0;
  for ( int j=nW-1; j; j--) {
    ( c |= CP[j] >> 32 ) <<= 32;
    ( c |= CP[j]       ) <<= 32;
  }
  ( c |= CP[0] >> 32 ) <<= 32;
  c |= CP[0];
  return c;
}

int rng::cpd( int ) {   // return CP degree
  return CPd;
}

// characteristic polynomial degree
const int rng::CPd = 19937;
```

```
// characteristic polynomial coefficients
const uz rng::CP[nW] = {
  0x0000000000000001ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0100000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000100000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000010ull , 0x0000000000000000ull , 0x0000000100000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0010000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000010000ull , 0x0000000000000000ull , 0x0000100000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000001ull ,
  0x0000000000000000ull , 0x0000000010000000ull , 0x0000000000000000ull , 0x0100000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0001000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000001000ull ,
  0x0000000000000000ull , 0x0000010000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000000010ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x1000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000001000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000010000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000100ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000001ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0080000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000100000ull , 0x0000000000000000ull , 0x0001a00000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x4000000000000000ull , 0x0000000000000010ull ,
  0x0000000000000000ull , 0x0000000124000000ull , 0x0000000000000000ull , 0x1050000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000001058000ull , 0x0000000000000000ull ,
  0x0000400000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000010480ull ,
  0x0000000000000000ull , 0x0000004100000000ull , 0x0000000000000000ull , 0x1800000000000000ull ,
  0x0000000000000104ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0008000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000010110000ull ,
  0x0000000000000000ull , 0x0001980000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000100004ull , 0x0000000000000000ull , 0x0001008860000000ull , 0x0000000000000000ull ,
  0x0400000000000000ull , 0x0000000000001001ull , 0x0000000000000000ull , 0x0000000018400000ull ,
  0x0000000000000000ull , 0x0000400000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000082600ull , 0x0000000000000000ull , 0x0001005000000000ull , 0x0000000000000000ull ,
  0x8000000000000000ull , 0x0000000001001805ull , 0x0000000000000000ull , 0x0000000040000000ull ,
  0x0000000000000000ull , 0x04a0000000000000ull , 0x0000000000010008ull , 0x0000000000000000ull ,
  0x0000000000400000ull , 0x0000000000000000ull , 0x0004000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000040ull , 0x0000000000000000ull , 0x0000022600000000ull ,
  0x0000000000000001ull , 0x4000000000000000ull , 0x0000000000000010ull , 0x0000000000000000ull ,
  0x0080000184000000ull , 0x0000000000000000ull , 0x0040000000000000ull , 0x0000000000000004ull ,
  0x0000000000000000ull , 0x0000a00060a40000ull , 0x0000000000000000ull , 0x0400400000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000400400ull , 0x0000000000000000ull ,
  0x4000404000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000024002624ull ,
  0x0000000000000000ull , 0x0050005040000000ull , 0x0000000000000000ull , 0x8400000000000000ull ,
  0x0000000000058005ull , 0x0000000000000000ull , 0x0000400040400000ull , 0x0000000000000000ull ,
  0x04a4000000000000ull , 0x0000000000000480ull , 0x0000000000000000ull , 0x0000004100404000ull ,
  0x0000000000000000ull , 0x1804040000000000ull , 0x0000000000000004ull , 0x0000000000000000ull ,
  0x0000000000000040ull , 0x0000000000000000ull , 0x0008022400000000ull , 0x0000000000000000ull ,
  0x4000000000000000ull , 0x0000000000110010ull , 0x0000000000000000ull , 0x0001980184000000ull ,
  0x0000000000000000ull , 0x0040000000000000ull , 0x0000000000000004ull , 0x0000000000000000ull ,
  0x0000008860a40000ull , 0x0000000000000000ull , 0x0400400000000000ull , 0x0000000000000001ull ,
  0x0000000000000000ull , 0x0000000018400400ull , 0x0000000000000000ull , 0x0000404000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000082624ull , 0x0000000000000000ull ,
  0x0001005040000000ull , 0x0000000000000000ull , 0x8400000000000000ull , 0x0000000000001805ull ,
  0x0000000000000000ull , 0x0000000040400000ull , 0x0000000000000000ull , 0x04a4000000000000ull ,
  0x0000000000000008ull , 0x0000000000000000ull , 0x0000000000404000ull , 0x0000000000000000ull ,
  0x0004040000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000040ull ,
  0x0000000000000000ull , 0x0000022400000000ull , 0x0000000000000000ull , 0x4000000000000000ull ,
  0x0000000000000010ull , 0x0000000000000000ull , 0x0000000184000000ull , 0x0000000000000000ull ,
  0x0040000000000000ull , 0x0000000000000004ull , 0x0000000000000000ull , 0x0000000060a40000ull ,
  0x0000000000000000ull , 0x0400400000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000400400ull , 0x0000000000000000ull , 0x0000404000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000002624ull , 0x0000000000000000ull , 0x0000005040000000ull ,
  0x0000000000000000ull , 0x8400000000000000ull , 0x0000000000000005ull , 0x0000000000000000ull ,
  0x0000000040400000ull , 0x0000000000000000ull , 0x04a4000000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000404000ull , 0x0000000000000000ull , 0x0004040000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000040ull , 0x0000000000000000ull ,
  0x0000022400000000ull , 0x0000000000000000ull , 0x4000000000000000ull , 0x0000000000000010ull ,
  0x0000000000000000ull , 0x0000000184000000ull , 0x0000000000000000ull , 0x0040000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000a40000ull , 0x0000000000000000ull ,
  0x0000400000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000400ull ,
  0x0000000000000000ull , 0x0000040000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000000024ull , 0x0000000000000000ull , 0x0000000040000000ull , 0x0000000000000000ull ,
  0x0400000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000400000ull ,
  0x0000000000000000ull , 0x0004000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull ,
  0x0000000000004000ull , 0x0000000000000000ull , 0x0000040000000000ull , 0x0000000000000000ull ,
  0x0000000000000000ull , 0x0000000000000000ull , 0x0000000000000000ull , 0x0000000200000000ull
} ;
```

Appendix C. Independent F_2-Linear RNG Implementation

This appendix appears in its original form, without editorial change.

INTENTIONALLY LEFT BLANK.

C.1 Makefile

```
## makefile

CF = -O2

CPP = g++

all:    jump_demo.exe jump_t113.exe jump_mt32.exe jump_t258.exe jump_mt64.exe

jump_%.exe: jump_gen.cpp rng_jump_%.h
        $(CPP) $(CF) -DRNG_JUMP_H=\"rng_jump_$*.h\" $< -o $@
        strip $@

clean:
        -rm -f *.o *.exe *~
```

C.2 Driver jump_gen.cpp

```
// -*-c++-*-
// jump_gen.cpp
// driver for rng class
// Random Number Generator with statistically independent instances

#include <iostream>
#include <sstream>
#include <iomanip>

using namespace std;

#include RNG_JUMP_H

#define USE() cout << "args are: -s seed -n number_of_rngs" << endl

int main( int argc , char* argv[] ) {
  int n_gen = 5;                // number of independent rngs

  // default seed
  unsigned seed=0x12340f00;

  if ( argc > 1 ) {
    stringstream ss("");
    for ( char** p=argv+1 ; *p ; ) ss << " " << *p++;
    string tok;
    while ( ! ss.eof() ) {
      ss >> tok;
      if      ( "-n"  == tok ) ss >> dec >> n_gen;
      else if ( "-s"  == tok ) ss >> hex >> seed;
      else { cout << "? " << tok << endl << USE(); exit(1); }
    }
  }

  // set the seed before constructing rngs
  rng::setseed( seed );

  // create rngs
  rng Z[n_gen];

  // call rngs and print
  for ( int i=0 ; i < n_gen ; i++ )
    cout << "call Z[" << dec << setw(2) << setfill (' ') << i << "].gen() = "
         << hex << setw(rng::wid) << setfill ('0') << Z[i].gen() << endl;

  // advance rngs to the same state
  cout << "synchronize the rngs" << endl;
  for ( int i=0 ; i < n_gen ; i++ )
    for ( int j = 0 ; j < (n_gen-i-1)*(1<<20) ;  j++ )
      Z[i].gen();

  // call rngs and print
  for ( int i=0 ; i < n_gen ; i++ )
    cout << "call Z[" << dec << setw(2) << setfill (' ') << i << "].gen() = "
         << hex << setw(rng::wid) << setfill ('0') << Z[i].gen() << endl;

  return 0;
}
```

C.3 Demonstration demo RNG class `rng_jump_demo.h`

```
// -*-c++-*-
// rng_jump_demo.h
// F2-Linear Feedback Shift Register Random Number Generator
// with statistically independent instances
#include <bitset>

class rng {
  struct state {                // state information
    unsigned int z;
  } ;
public:
  // the first instance is initialized from the seed
  // subsequent instances are jumped ahead giving independent rngs
  rng( );                       // constructor (called with no arguments)
  rng( const unsigned& seed );  // constructor (called with seed argument)
  unsigned gen( );              // rng returns a 32-bit unsigned integer
  static void setseed( const unsigned& seed ); // call once for all rngs
  static const int wid = 8;     // integer format width
private:
  state s ;                     // rng state
  unsigned id ;                 // rng sequential id per instance
  void init( );                 // initialize a rng
  void seedgen( );              // seed the first rng
  void jump( );                 // jump to new state
  void jpunpack( const unsigned& Jump_P );
  static unsigned seed;         // seed for first generator
  static unsigned ref_counter;  // number of instances
  static state ref_state;       // zero state of newest instance
  static unsigned Jump_P[];     // jump polynomial coefficients
  static bitset<32> JP;         // jump polynomial
};

rng::rng( ) {                   // constructor (called with no arguments)
  init( );
}

rng::rng( const unsigned& seed ) { // constructor (called with seed argument)
  setseed( seed );
  init( );
}

void rng::setseed( const unsigned& seed ) {
  if ( ref_counter == 0 )
    rng::seed = seed;
}

unsigned rng::gen( ) {          // rng returns a 32-bit unsigned integer
  // this is Tausworthe t088.0 (the first generator), period = 2^31-1
  s.z = ( (s.z & 0xfffffffe) << 12 ) ^ ( ( (s.z << 13) ^ s.z ) >> 19 );
  return s.z;
}

void rng::init( ) {
  // the first rng state s is seeded as indicated
  // for subsequent rngs compute state s
  // jumped ahead from the reference state
  // set reference state for next jump to zero state of current rng
  if ( ref_counter == 0 ) {
    seedgen( );
    jpunpack( Jump_P[0] );
  }
  else
    jump( );
  ref_state = s ;
  id = ref_counter++;
}

void rng::seedgen( ) {
  // seed the system
  // set seed for first generator (algorithm specific to rng)
  // the rng must be in state zero, not the seed state
  s.z = seed;
  gen( );
}

void rng::jpunpack( const unsigned& Jump_P ) {
  JP |= Jump_P;
}
void rng::jump( ) {
  // set rng state s to reference state
  // compute temp_state = J[0]*s[0] + ... + J[31]*s[31]
  // set s to temp_state
  // now s is jumped ahead from reference state by J
  state temp_state;
  s = ref_state;
  temp_state.z = 0;
  for ( int i = 0 ; i < 32 ; i++ ) {
    if ( JP[i] )
      temp_state.z ^= s.z;
    gen( );
  }
  s = temp_state ;
}

// number of rngs instantiated
unsigned rng::ref_counter=0;

// reference state for jump computation is the
// zero state of the previous instance
rng::state rng::ref_state;

// seed for first rng instance
unsigned rng::seed=0x12340f00;

// Jump polynomial
bitset<32> rng::JP = 0;

// jump polynomial coefficients for jump = 2^20
unsigned rng::Jump_P[1] = {
  0x4fbc5320
};
```

C.4 Tausworthe t113 RNG class rng_jump_t113.h

```
// -*-c++-*-
// rng_jump_t113.h
// Tausworthe Random Number Generator
// with statistically independent instances
#include <cstring>
#include <bitset>

class rng {
  struct state {                // state information
    unsigned int z[4];
  } ;
public:
  // the first instance is initialized from the seed
  // subsequent instances are jumped ahead giving independent rngs
  rng( );                       // constructor (called with no arguments)
  rng( const unsigned& seed );  // constructor (called with seed argument)
  unsigned gen( );              // rng returns a 32-bit unsigned integer
  static void setseed( const unsigned& seed ); // call once for all rngs
  static const int wid = 8;     // integer format width
private:
  state s ;                     // rng state
  unsigned id ;                 // rng sequential id per instance
  void init( );                 // initialize a rng
  void seedgen( );              // seed the first rng
  void jump( );                 // jump to new state
  void jpunpack( const unsigned* const Jump_P );
  static unsigned seed;         // seed for first generator
  static unsigned ref_counter;  // number of instances
  static state ref_state;       // zero state of newest instance
  static unsigned Jump_P[][4];  // jump polynomial coefficients
  static bitset<32> JP[4];      // jump polynomial
};


rng::rng( ) {                   // constructor (called with no arguments)
  init( );
}

rng::rng( const unsigned& seed ) { // constructor (called with seed argument)
  setseed( seed );
  init( );
}

void rng::setseed( const unsigned& seed ) {
  if ( ref_counter == 0 )
    rng::seed = seed;
}

unsigned rng::gen( ) {          // rng returns a 32-bit unsigned integer
  s.z[0] = ((s.z[0] & 0xfffffffe) << 18) ^ (((s.z[0] <<  6) ^ s.z[0]) >> 13);
  s.z[1] = ((s.z[1] & 0xfffffff8) <<  2) ^ (((s.z[1] <<  2) ^ s.z[1]) >> 27);
  s.z[2] = ((s.z[2] & 0xfffffff0) <<  7) ^ (((s.z[2] << 13) ^ s.z[2]) >> 21);
  s.z[3] = ((s.z[3] & 0xffffff80) << 13) ^ (((s.z[3] <<  3) ^ s.z[3]) >> 12);
  return ( s.z[0] ^ s.z[1] ^ s.z[2]^ s.z[3] ) ;
}

void rng::init( ) {
  // the first rng state s is seeded as indicated
  // for subsequent rngs compute state s
  // jumped ahead from the reference state
  // set reference state for next jump to zero state of current rng
  if ( ref_counter == 0 ) {
    seedgen( );
    jpunpack( Jump_P[0] );
  }
  else
    jump( );
  ref_state = s ;
  id = ref_counter++;
}

void rng::seedgen( ) {
  // seed the system
  // set seed for first generator (algorithm specific to rng)
  // the rng must be in state zero, not the seed state
  s.z[0] = seed   * 69069;
  if ( s.z[0] <   2 ) s.z[0] += 2U;
  s.z[1] = s.z[0] * 69069;
  if ( s.z[1] <   8 ) s.z[1] += 8U;
  s.z[2] = s.z[1] * 69069;
  if ( s.z[2] <  16 ) s.z[2] += 16U;
  s.z[3] = s.z[2] * 69069;
  if ( s.z[3] < 128 ) s.z[3] += 128U;
  gen();
}

void rng::jpunpack( const unsigned* const Jump_P ) {
  for ( int j=0; j<4; j++ )
    JP[j] |= Jump_P[j];
}

void rng::jump( ) {
  // set rng state s to reference state
  // compute temp_state = J[0]*s[0] + ... + J[31]*s[31]
  // set s to temp_state
  // now s is jumped ahead from reference state by J
  int i , j;
  state temp_state;
  s = ref_state;
  memset( temp_state.z , 0 , sizeof(temp_state.z) );
  for ( i = 0 ; i < 32 ; i++ ) {
    for ( j = 0 ; j < 4 ; j++ )
      if ( JP[j][i] )
        temp_state.z[j] ^= s.z[j];
    gen();
  }
  s = temp_state ;
}

// number of rngs instantiated
unsigned rng::ref_counter=0;

// reference state for jump computation is the
// zero state of the previous instance
rng::state rng::ref_state;

// seed for first rng instance
unsigned rng::seed=0x12340f00;

// Jump polynomial
bitset<32> rng::JP[4] = { 0, 0, 0, 0 };
```

```
// Taus113 jump coefficients
unsigned int rng::Jump_P[1][4] = {
  {
    // 2^20
    0x0c382e31 , 0x1b040425 , 0x0b49a509 , 0x0173f6b0

    // 2^80
    // 0x487cf69c , 0x00be6310 , 0x04bfe2bb , 0x000824f9
  }
};
```

C.5 Tausworthe `t258` RNG class `rng_jump_t258.h`

```
// -*-c++-*-
// rng_jump_t258.h
// Tausworthe Random Number Generator
// with statistically independent instances
#include <cstring>
#include <bitset>

class rng {
  struct state {                // state information
    unsigned long long int z[5];
  } ;
public:
  // the first instance is initialized from the seed
  // subsequent instances are jumped ahead giving independent rngs
  rng( );                       // constructor (called with no arguments)
  rng( const unsigned long long int& seed );    // constructor (called with seed argument)
  unsigned long long int gen( ); // rng returns a 64-bit unsigned integer
  static void setseed( const unsigned long long int& seed ); // call once for all rngs
  static const int wid = 16;    // integer format width
private:
  state s ;                     // rng state
  unsigned id ;                 // rng sequential id per instance
  void init( );                 // initialize a rng
  void seedgen( );              // seed the first rng
  void jump( );                 // jump to new state
  void jpunpack( const unsigned long long* const Jump_P );
  static unsigned long long int seed; // seed for first generator
  static unsigned ref_counter;  // number of instances
  static state ref_state;       // zero state of newest instance
  static unsigned long long Jump_P[][5];  // jump polynomial coefficients
  static bitset<64> JP[5];      // jump polynomial
};

rng::rng( ) {                   // constructor (called with no arguments)
  init( );
}

rng::rng( const unsigned long long& seed ) { // constructor (called with seed argument)
  setseed( seed );
  init( );
}

void rng::setseed( const unsigned long long& seed ) {
  if ( ref_counter == 0 )
    rng::seed = seed;
}

unsigned long long rng::gen( ) { // rng returns a 64-bit unsigned integer
  s.z[0] = ((s.z[0] & 0xfffffffffffffffeull) << 10) ^ (((s.z[0] <<  1) ^ s.z[0]) >> 53);
  s.z[1] = ((s.z[1] & 0xfffffffffffffe00ull) <<  5) ^ (((s.z[1] << 24) ^ s.z[1]) >> 50);
  s.z[2] = ((s.z[2] & 0xfffffffffffff000ull) << 29) ^ (((s.z[2] <<  3) ^ s.z[2]) >> 23);
  s.z[3] = ((s.z[3] & 0xfffffffffffe0000ull) << 23) ^ (((s.z[3] <<  5) ^ s.z[3]) >> 24);
  s.z[4] = ((s.z[4] & 0xffffffffff800000ull) <<  8) ^ (((s.z[4] <<  3) ^ s.z[4]) >> 33);
  return ( s.z[0] ^ s.z[1] ^ s.z[2] ^ s.z[3] ^ s.z[4] ) ;
}

void rng::init( ) {
  // the first rng state s is seeded as indicated
  // for subsequent rngs compute state s
  // jumped ahead from the reference state
  // set reference state for next jump to zero state of current rng

  if ( ref_counter == 0 ) {
    seedgen( );
    jpunpack( Jump_P[0] );
  }
  else
    jump( );
  ref_state = s ;
  id = ref_counter++;
}

void rng::seedgen( ) {
  // seed the system
  // set seed for first generator (algorithm specific to rng)
  // the rng must be in state zero, not the seed state
  s.z[0] = seed * 69069;
  if ( s.z[0] < 2 ) s.z[0] += 2u;
  s.z[1] = s.z[0] * 69069;
  if ( s.z[1] < 8 ) s.z[1] += 8u;
  s.z[2] = s.z[1] * 69069;
  if ( s.z[2] < 16 ) s.z[2] += 16u;
  s.z[3] = s.z[2] * 69069;
  if ( s.z[3] < 128 ) s.z[3] += 128u;
  s.z[4] = s.z[3] * 69069;
  if ( s.z[4] < 2048 ) s.z[4] += 2048u;
  gen();
}

void rng::jpunpack( const unsigned long long* const Jump_P ) {
  for ( int j = 0 ; j < 5 ; j++ ) {
    JP[j] |= unsigned( Jump_P[j] >> 32 & 0xffffffff );
    JP[j] <<= 32;
    JP[j] |= unsigned( Jump_P[j]       & 0xffffffff ) ;
  }
}

void rng::jump( ) {
  // set rng state s to reference state
  // compute temp_state = J[0]*s[0] + ... + J[63]*s[63]
  // set s to temp_state
  // now s is jumped ahead from reference state by J
  int i , j;
  state temp_state;
  s = ref_state;
  memset( temp_state.z , 0 , sizeof(temp_state.z) );
  for ( i = 0 ; i < 64 ; i++ ) {
    for ( j = 0 ; j < 5 ; j++ )
      if ( JP[j][i] )
        temp_state.z[j] ^= s.z[j];
    gen();
  }
  s = temp_state ;
}

// number of rngs instantiated
unsigned rng::ref_counter=0;

// reference state for jump computation is the
// zero state of the previous instance
rng::state rng::ref_state;
```

```
// seed for first rng instance
unsigned long long rng::seed=0x12340f00;

// Jump polynomial
bitset<64> rng::JP[5] = { 0, 0, 0, 0, 0 };

// Taus113 jump coefficients
unsigned long long rng::Jump_P[1][5] = {
  {
    // 2^20
    0x7fd7f8ffc40b6103ull , 0x0008a943bd59e7bfull , 0x000b0e2f0e8a5511ull ,
    0x000050f342c7f5ddull , 0x000000d9bd741142ull

    // 2^80
    // 0x702fffbefc7e2115ull , 0x007f0672769e600cull , 0x000aa7dad2018eedull ,
    // 0x0000598f11394622ull , 0x0000000080001002ull

    // 2^128
    // 0x0000000000000010ull , 0x0057eb3124fac097ull , 0x000e88c757d05b63ull ,
    // 0x00005f36f89389baull , 0x0000000100000000ull
  }
};
```

C.6 Mersenne Twister mt32 RNG class rng_jump_mt32.h

```
// -*-c++-*-
// rng_jump_mt32.h
// Mersenne Twister Random Number Generator
// with statistically independent instances
#include <cstring>
#include <bitset>

class rng {
  static const unsigned N = 624;
  struct state {                // state information
    unsigned n;
    unsigned y;
    unsigned z[N];
  } ;
public:
  // the first instance is initialized from the seed
  // subsequent instances are jumped ahead giving independent rngs
  rng( );                       // constructor (called with no arguments)
  rng( const unsigned& seed );  // constructor (called with seed argument)
  unsigned gen( );              // rng returns a 32-bit unsigned integer
  static void setseed( const unsigned& seed ); // call once for all rngs
  static const int wid = 8;     // integer format width
private:
  state s ;                     // rng state
  unsigned id ;                 // rng sequential id per instance
  void init( );                 // initialize a rng
  void seedgen( );              // seed the first rng
  void jump( );                 // jump to new state
  void jpunpack( const unsigned* const Jump_P );
  static unsigned seed;         // seed for first generator
  static unsigned ref_counter;  // number of instances
  static state ref_state;       // zero state of newest instance
  static unsigned Jump_P[][N];  // jump polynomial coefficients
  static const unsigned M = 397;
  static const unsigned MATRIX_A   = 0x9908b0df; // constant vector a
  static const unsigned UPPER_MASK = 0x80000000; // most significant w-r bits
  static const unsigned LOWER_MASK = 0x7fffffff; // least significant r bits
  static const unsigned CPdeg = 19937; // characteristic polynomial degree
  static const unsigned Jdmax = CPdeg-1; // max jump polynomial degree
  static bitset<CPdeg> JP;      // jump polynomial
};

rng::rng( ) {                   // constructor (called with no arguments)
  init( );
}

rng::rng( const unsigned& seed ) { // constructor (called with seed argument)
  setseed( seed );
  init( );
}

void rng::setseed( const unsigned& seed ) {
  if ( ref_counter == 0 )
    rng::seed = seed;
}

unsigned rng::gen( ) {          // rng returns a 32-bit unsigned integer
  unsigned y;
  static unsigned mag01[2]={0x0, MATRIX_A};
  if (s.n >= N) {               // generate N words at one time
```

```
    int kk;
    for (kk=0;kk<N-M;kk++) {
      s.y = (s.z[kk]&UPPER_MASK)|(s.z[kk+1]&LOWER_MASK);
      s.z[kk] = s.z[kk+M] ^ (s.y >> 1) ^ mag01[s.y & 0x1];
    }
    for (;kk<N-1;kk++) {
      s.y = (s.z[kk]&UPPER_MASK)|(s.z[kk+1]&LOWER_MASK);
      s.z[kk] = s.z[kk+(M-N)] ^ (s.y >> 1) ^ mag01[s.y & 0x1];
    }
    s.y = (s.z[N-1]&UPPER_MASK)|(s.z[0]&LOWER_MASK);
    s.z[N-1] = s.z[M-1] ^ (s.y >> 1) ^ mag01[s.y & 0x1];
    s.n = 0;
  }
  y = s.y = s.z[s.n++];
  // Tempering
  y ^= (y >> 11);
  y ^= (y <<  7) & 0x9d2c5680;
  y ^= (y << 15) & 0xefc60000;
  y ^= (y >> 18);
  return y;
}

void rng::init( ) {
  // the first rng state s is seeded as indicated
  // for subsequent rngs compute state s
  // jumped ahead from the reference state
  // set reference state for next jump to zero state of current rng
  if ( ref_counter == 0 ) {
    seedgen( );
    jpunpack( Jump_P[0] );
  }
  else
    jump( );
  ref_state = s ;
  id = ref_counter++;
}

void rng::seedgen( ) {
  // seed the system
  // set seed for first generator (algorithm specific to rng)
  // the rng must be in state zero, not the seed state
  s.z[0]= seed & 0xffffffff;
  for (s.n=1; s.n<N; s.n++) {
    s.z[s.n] = (1812433253 * (s.z[s.n-1] ^ (s.z[s.n-1] >> 30)) + s.n);
    s.z[s.n] &= 0xffffffff;
  }
  gen();                        // for all rngs
}

void rng::jpunpack( const unsigned* const Jump_P ) {
      for ( int j=N-1; j; j-- )
        ( JP |= Jump_P[j] ) <<= 32;
      JP |= Jump_P[0];
}

void rng::jump( ) {
  // compute jump state t.z[0 .. (N-1)]
  // from jump polynomial J and base states sy=s.y
  //
  // t.z[0  ] = J[0]*sy[0  ] + .. + J[d]*sy[d    ] + .. + J[Jdmax]*sy[Jdmax    ]
  // t.z[i  ] = J[0]*sy[i  ] + .. + J[d]*sy[d+i  ] + .. + J[Jdmax]*sy[Jdmax+i  ]
  // t.z[N-1] = J[0]*sy[N-1] + .. + J[d]*sy[d+N-1] + .. + J[Jdmax]*sy[Jdmax+N-1]
  state t;
  unsigned sy[Jdmax+N]; // Jdmax+N states 0 .. (Jdmax+N-1) for jump
  s = ref_state;                // start s at ref state
  for ( int i=0 ; i<Jdmax+N ; i++ ) {
    sy[i] = s.y ;               // save states sy[i in 0 .. (Jdmax+N-1)]
    gen() ;
  }
  memset( t.z , 0 , sizeof(t.z) );
  // J term power = d in 0 .. Jdmax
  int i, k;
  for ( int d=0 ; d<=Jdmax ; d++ )
    if ( JP[d] == 1 )
      // i in 0 .. (N-1)
      // k in d .. (d+N-1) for each d
      // k in 0 .. (Jdmax+N-1) overall
      for ( i=0 , k=d ; i<N ; i++ , k++ )
        t.z[i] ^= sy[k] ;       // k = d + i
  s = t ;
  s.y = s.z[0];
  s.n = 1;
}

// number of rngs instantiated
unsigned rng::ref_counter=0;

// reference state for jump computation is the
// zero state of the previous instance
rng::state rng::ref_state;

// seed for first rng instance
unsigned rng::seed=0x12340f00;

// Jump polynomial
bitset<rng::CPdeg> rng::JP = 0;

// MT32 jump coefficients
unsigned rng::Jump_P[1][N] = { // jump polynomial
  {
```

```
// 2^20
0x6df32afe , 0x7eab0cac , 0xbefa4d71 , 0xadc4c8b2 , 0x23f269f1 , 0xf0955951 , 0xa19bc77c , 0xccf01d2f ,
0x89f90a3b , 0x67691fcd , 0xc612f21b , 0x58846827 , 0xea84d6f9 , 0xa024724a , 0x11e0ae46 , 0x7f1c4ca3 ,
0x93e769bc , 0x60683f55 , 0xf86e41bc , 0x5d567f3c , 0x2e1f091d , 0x692a7e93 , 0x2359ef12 , 0x0b58b2c0 ,
0xf06d9493 , 0xeccdef01 , 0x7e2be6a2 , 0x3cf24c13 , 0x1c9f44db , 0x3fb5d8a4 , 0x144d3faf , 0x320092e9 ,
0x7054523c , 0x42997177 , 0x1787d509 , 0xdebc078a , 0x5aaeb386 , 0x931fd6a6 , 0xa7b8ddfe , 0xe03e657b ,
0x0f9920b3 , 0x7b9905ca , 0xc6b0c6c1 , 0x4dd5202c , 0x40359cf1 , 0x32e3cb72 , 0xa8030761 , 0x5c433f85 ,
0x4af627c4 , 0x2690772c , 0x735adb08 , 0x2ee6afdc , 0x7a5a6c81 , 0x0898faf7 , 0xbaf1c4a2 , 0xd62a8fea ,
0xfb14df97 , 0x0c5144c2 , 0x3f16b27e , 0xc18df4b3 , 0x6eddc0cf , 0xad5a221c , 0x5145bed4 , 0xad742fe2 ,
0xaeb3370b , 0x4fe7a88b , 0x03d8bfd0 , 0xbb7da76e , 0xd37ba3d4 , 0x4f0b4778 , 0x0931d190 , 0xc4c93cbd ,
0x7361dc46 , 0x6a357040 , 0xad3442f4 , 0xcf778ab0 , 0x082d3bfe , 0x12915fe6 , 0xf1a38cb1 , 0xf069cd80 ,
0xcb8e1763 , 0x02d67f85 , 0xb19693f8 , 0xc4c7246b , 0xc6ccda01 , 0x1e7a413b , 0xbd606660 , 0xa18a553a ,
0xfecc9766 , 0x5963723d , 0x3cb73c9b , 0xae9efa7b , 0xf4e84c1f , 0xf2bb02aa , 0x3a0a1227 , 0xeefe6b5a ,
0xeb2ffe48 , 0x21f46276 , 0xd0b930df , 0x45657132 , 0xb11ecaa4 , 0xcaa88258 , 0x82dae28f , 0xf7de8900 ,
0x2bb6d4d8 , 0x83a40bae , 0x1e48a137 , 0x64cd3bc7 , 0x1d79c98d , 0x20fda111 , 0xe8f90e31 , 0xc389e322 ,
0xe9491785 , 0xb09b966f , 0xf6c96f54 , 0x923a8c0d , 0x47e1c778 , 0x53348054 , 0xe3f7fb74 , 0x510245ca ,
0xfcd8ee40 , 0xf33d907c , 0xc893f8ae , 0x12486ee1 , 0x61de23f5 , 0x2c50648b , 0x7a50d21f , 0x913b6d92 ,
0x194b0bdc , 0x9aa1f5d0 , 0x59b7fbc5 , 0x1563fdbc , 0xa0b8c39c , 0x84ceeaba , 0x24045370 , 0x4c70236c ,
0x76d8abce , 0x7518f2f4 , 0x54d41d16 , 0x9c6e661d , 0x09cf99fc , 0x62967b52 , 0xfee508b1 , 0x2a734b9f ,
0x50135637 , 0x27f8726c , 0xa8d64246 , 0x0d79c054 , 0xe135ce55 , 0x8ca16ff9 , 0x661a80ef , 0x1aed170c ,
0xaa02cbac , 0xc02e1168 , 0x52722adc , 0x4f5a2505 , 0x522b22f9 , 0xac4f4300 , 0xb974f1e9 , 0xc8e41ae8 ,
0x1d558c63 , 0x29f0cab4 , 0x2370640b , 0xa6e063f7 , 0x34e436e7 , 0xcefc457a , 0xa7107b46 , 0x2056d593 ,
0x60acef1f , 0xcc302c9f , 0xd8180b7a , 0xa8b82496 , 0xc81e5237 , 0x83671917 , 0xb17db7bd , 0xa82f8b02 ,
0xc143f425 , 0x023c3573 , 0x9ec7da44 , 0xbc47fffe , 0xd37b9ed1 , 0x2a9e7eb0 , 0xaddceca4 , 0x91cad071 ,
0x4d17cc60 , 0xeab1107c , 0x6403cc01 , 0x3d868f19 , 0x5be5a795 , 0x7a3aa33e , 0xe239e614 , 0x8c84cff5 ,
0x708e558d , 0x35c52964 , 0xb0fb49eb , 0xfb059a02 , 0xc74e00d9 , 0xa25e1c8d , 0x3e33c11e , 0x4baff421 ,
0x1987e12c , 0xe4d54472 , 0x453237dc , 0xb68937df , 0xd7878393 , 0x7c08fe2e , 0xd23d2a0b , 0x90fb33ec ,
0xe6e6c8fd , 0x09281233 , 0x62571b02 , 0xf08df485 , 0x0956b5d9 , 0xa4f3f91c , 0x46a631ef , 0x7cbe51e1 ,
0xc7857b86 , 0xed67e22e , 0xb8b7dac0 , 0x017fb4e6 , 0x4b9803e5 , 0xb57b6432 , 0x30f37aa6 , 0xf4fc73d3 ,
0x4866c355 , 0x9ec898fa , 0xeca55312 , 0xd8156db6 , 0xa5ba8807 , 0x4dcc4554 , 0x799e57d3 , 0xd3413725 ,
0x45662187 , 0x7b2dff59 , 0x7d3c8071 , 0xc79def33 , 0x161a09fa , 0xdfbf1fb2 , 0x914d5f1b , 0x86021f42 ,
0x4b82806a , 0x2bac0545 , 0x8ab601f3 , 0xc00220bd , 0xc482d0a5 , 0xa76f5824 , 0xf52bb5f3 , 0xe0f55c25 ,
0x47f25d13 , 0x42f71ffb , 0x09dfafd6 , 0x6b93afc7 , 0xbe8aeb01 , 0x79bc0d1f , 0x4e843718 , 0x5c8c13fc ,
0x141a6e5c , 0x55bfcdd6 , 0xd2fcf28a , 0x14ee8ed7 , 0x1c93d772 , 0xc129008e , 0x61805675 , 0x27e22987 ,
0xb2aae41a , 0xbad5b1db , 0xe5c7e0d9 , 0xffe7b659 , 0xbae4c34e , 0xe3a22059 , 0xa51f6fa7 , 0x05816c33 ,
0xe61c276d , 0xd92d83cd , 0xcfcd327f , 0xb083cd35 , 0x3821f57e , 0x372ca022 , 0xbd003b47 , 0x0dc1aa8d ,
0x3897089d , 0xba148634 , 0xb3d53f2a , 0x480b174c , 0xe091d5f3 , 0x7589c8bb , 0x15cedae0 , 0xc42d80e6 ,
0xd5cc304c , 0xb810d6ea , 0x035cb0d4 , 0x03226722 , 0x6796e622 , 0x2882eb50 , 0xc8b4968c , 0x6dc5aab8 ,
0x6bed6442 , 0xc0371b44 , 0x35f273c4 , 0xdc855590 , 0xff734de5 , 0x20840dea , 0x5f9df6c7 , 0xfa4e1cd5 ,
0x9cc64e10 , 0xd874afa1 , 0x198d2a62 , 0x13d92dcf , 0x938539c0 , 0xce691092 , 0xbf2dced7 , 0xf043abdd ,
0xc2568953 , 0xdeea32ed , 0xb8346176 , 0xbdca4be9 , 0xfe0ba665 , 0x3392e1a4 , 0xb6d7ee0e , 0xe7287f2e ,
0x6eac7044 , 0x73211586 , 0x1644b661 , 0x64914aa8 , 0x38fc5881 , 0xec3f5a1e , 0x33782633 , 0x8f3ccff3 ,
0x130acc82 , 0xe16adee2 , 0x083824e4 , 0x2adef6ba , 0x66864277 , 0x00e7ba58 , 0xf4bf11b9 , 0x82a9435b ,
0x993ae794 , 0xd1da7674 , 0x85b818fa , 0x3fb184c4 , 0x9d447a17 , 0x7f4f38ac , 0x33aad744 , 0xb744aa59 ,
0x94a0e6a2 , 0x5b877e7e , 0x333473ad , 0x4099a4fe , 0xffe05fc8 , 0x3148b0f2 , 0x03f4d5bc , 0xb7166e53 ,
0x038ca301 , 0x025b101a , 0x691d2d45 , 0x2c79fe2e , 0xf8670882 , 0xdd7f137e , 0x648ec02b , 0x2450138c ,
0xc7faf2d9 , 0xf9eaa8d4 , 0x5eb7e2fd , 0x8598ba54 , 0x45056d6d , 0x2049d7a3 , 0x0458e03b , 0xcc729790 ,
0x9a6c4eb6 , 0x1f1199e6 , 0x8270379f , 0xc2904f08 , 0xde7a97ea , 0x6ae6b57a , 0xe680ce9d , 0xa869aa4c ,
0x91ed8767 , 0x2a85b8d7 , 0x53cfa372 , 0x56d07626 , 0x662ec41b , 0x3b8b74b7 , 0x0d18bea3 , 0x2c2f19f0 ,
0xa6d6b0a9 , 0xf0a8e819 , 0x1e0aba84 , 0xadbb42d8 , 0x7e64ae3a , 0x049bc863 , 0xbebfa1e4 , 0x0c110c39 ,
0x38f1eb3c , 0xa2aec7bf , 0xd2e73d92 , 0x79a525bd , 0x9f8e4e0f , 0x266d0ca2 , 0xb3f8b9fb , 0x12cb1000 ,
0xbb56ac97 , 0xd90189f2 , 0x5aa6591c , 0x92485e89 , 0x645321ac , 0xa92743e7 , 0x811d2e95 , 0x19df1d2b ,
0x80f64590 , 0xbd7c8a75 , 0x66562d6d , 0xacb28d28 , 0xce1fe1e7 , 0x7a138a01 , 0xd63ea468 , 0x1b4497a5 ,
0x85f1f0b2 , 0x7c99b97e , 0x4b0113e3 , 0x5628bafd , 0x8ba7e042 , 0x9ab5d848 , 0xe3e6b974 , 0x77932da8 ,
0x40f7e374 , 0xc9b6bc5e , 0x8f14a77e , 0x6e8f22ea , 0x4bf2d014 , 0xab61d46d , 0xd0d43940 , 0xe28d074b ,
0xc854c216 , 0xdba749dd , 0x997737af , 0x12651c8a , 0x82245826 , 0x367ed190 , 0x698468d4 , 0xf4bb2137 ,
0x8fe114f5 , 0x0596b440 , 0x678e7f4f , 0x15866f46 , 0x3bb4e9f9 , 0x550272a2 , 0x1cb36315 , 0xc9baa847 ,
0xfab56ce1 , 0xdaa587b5 , 0xd8d34788 , 0xac166b9b , 0x24f64548 , 0x24b1ed72 , 0xd0daff29 , 0x4a52ea11 ,
0x63f809ed , 0x061bd7c3 , 0x2d0679e2 , 0x58d2476e , 0x34026f04 , 0x686c948f , 0x099e8172 , 0x41902863 ,
0xd3efa08c , 0xc22ce857 , 0xf1ac125c , 0x97b49fe6 , 0x12a8107b , 0x387f3602 , 0x554673ba , 0x02442fae ,
0x6ff00875 , 0xc4b19427 , 0xe2c08c3c , 0xe9bf1881 , 0x90580b2f , 0xc58d42b8 , 0x21c93ca5 , 0x22ac3484 ,
0x8e377962 , 0x7eefc4a3 , 0x7275f6e7 , 0xe9c9f32b , 0xfd2c46db , 0x63eb94fb , 0x5073a894 , 0x8750e968 ,
0xed0e7c75 , 0xc2ded347 , 0x9e3d0dc8 , 0x498271fb , 0x70ff97ca , 0x4aab151b , 0x7218d914 , 0x4d010e68 ,
0x4689d046 , 0x2a707427 , 0xbe623eed , 0xfed455ff , 0x53690eac , 0xb848310d , 0xf3fcb7b0 , 0x40dc3283 ,
0xf764e416 , 0xb1fd7da7 , 0xd024e345 , 0xedbb9dd2 , 0x79c99b3c , 0x13b0e5e5 , 0x51ca5dd3 , 0xdae497e3 ,
0x2256090e , 0xf1295c04 , 0xe14f4c15 , 0xd0a26a57 , 0x0ffcaa12 , 0x83b8c4f2 , 0xe4007c96 , 0xa7a6a41b ,
0x8205a30b , 0x14b2afa5 , 0xdac6904d , 0xe5ce10d8 , 0xcbbf27b8 , 0x4fb7d01c , 0xd53bc9b8 , 0x02d9ce36 ,
0xacf26d97 , 0x703de1f9 , 0x02cc9618 , 0x45984d87 , 0xf558eb3f , 0x05ef6841 , 0x88be1ee7 , 0x39a2703f ,
0xd97ca04e , 0x48e35039 , 0x0cda8881 , 0xe68481ef , 0x25921fd0 , 0x9c6be6d5 , 0x38b79457 , 0x54994e56 ,
0x333b4e9f , 0xd9ea551e , 0x060c1cfe , 0x9101e16c , 0x1081af6c , 0xd2d2dd81 , 0x3bc080bb , 0xb3fef358 ,
0x476b987b , 0x62dee751 , 0xb8c2e9af , 0x3209e7e2 , 0x719e9b4a , 0x53615727 , 0xa85aa982 , 0xe79f0c32 ,
0xc568e2bf , 0x636ade37 , 0xb80eb8a0 , 0x82ba431a , 0x272de181 , 0xafd623c8 , 0x1ac6475a , 0x2c6beaee ,
0xa20ee8cd , 0x13735ce5 , 0xee967dd4 , 0x21d58e76 , 0x65a7d016 , 0x08b6e7b0 , 0x631e38d0 , 0x8d1639a9 ,
0x54435fc7 , 0xdf32cf3a , 0x4a3d022c , 0x4c40e3e4 , 0x124b242e , 0x14d1aade , 0xffa63459 , 0x4f3c0062 ,
0x4d8f8d6e , 0x8e42994d , 0x73bb383c , 0x3d7f9110 , 0xc399e168 , 0xc6a1b279 , 0x3771ba04 , 0xd533f333 ,
0x4eec9510 , 0xb0571327 , 0xacea1b64 , 0xdb9baa85 , 0xa327e725 , 0xaa682a2b , 0x8efc4ad8 , 0xcae6bdc3 ,
0x74110fe6 , 0xcddbcada , 0x7c11a9cd , 0xecdf108a , 0x64c83e51 , 0xe66ccf5f , 0x6353efd9 , 0xf98d6ccd ,
0xce5894bb , 0x0151b56c , 0xde8117cf , 0xc1690409 , 0x7d282b3c , 0xcd48ee59 , 0xd6002857 , 0xf12318f3 ,
0x9639130f , 0xaabb9e4d , 0x8c5650d9 , 0x9c1a506b , 0xde395cb0 , 0x38841ec8 , 0x43360adc , 0x00000001
```

```
    // 2^10000
    // 0xc04347b3 , 0x568188e5 , 0xf6c1d4d0 , 0xdd9ff6f9 , 0x1e07ec3a , 0x27c9cd71 , 0xdb5f3e52 , 0x8fd2c779 ,
    // 0xf33d1f0f , 0x350a5ea5 , 0x4b098eb2 , 0x9c2e5c2f , 0x2cf19af8 , 0xf2515f40 , 0x8fb9cfa0 , 0xfaa5a36c ,
    // 0x7a47cd4a , 0xb22e9739 , 0x8e340824 , 0xb515d2bf , 0x716e3103 , 0xda0a315f , 0xfcc1cce4 , 0x4d20c264 ,
    // 0xf9d6bc5b , 0xea00f60c , 0xdfd13eee , 0x94a9d212 , 0x85fb2682 , 0xe8a14ffd , 0x8160533c , 0x512c3ec3 ,
    // 0x40df671c , 0xcd5b5a7e , 0xd76a8f57 , 0xeb84a72b , 0x3e71f910 , 0xe44f8190 , 0x5521bb0c , 0xb7bb65b1 ,
    // 0x7c8a942e , 0xbd704511 , 0x5e15e781 , 0x12e3cc1e , 0xa736a0e3 , 0x437998a9 , 0xaf362081 , 0x42a8e62b ,
    // 0x0033ad3e , 0x54deda1c , 0x4f05ed13 , 0x2a0a7124 , 0xa19449ba , 0x45030aad , 0x5c84d8d7 , 0xcf60fd1a ,
    // 0x891db7cf , 0x96c26ee3 , 0xaca90ac8 , 0xa71fa59e , 0xe631c7b3 , 0x387e1b29 , 0x2e02a376 , 0xb98cc62c ,
    // 0x790624f4 , 0xc1e64a2e , 0xf1660529 , 0x595e5fbe , 0xc334e147 , 0x402eb5b1 , 0x7ceacc2f , 0x3666b776 ,
    // 0xa3d1f2c5 , 0x22603362 , 0x71361ed6 , 0xaa2df833 , 0x8636a3b6 , 0x7c711139 , 0xee71425b , 0x631ed752 ,
    // 0x7c213109 , 0x937e010c , 0x112f65ec , 0x2b94197a , 0xcb6944b6 , 0xb2ea7012 , 0xcaa0edd4 , 0x1aa56252 ,
    // 0x2aba1869 , 0x052a69c5 , 0x270f8cda , 0x4519f6ee , 0x0de03259 , 0x66687b7e , 0xa2b6fa7b , 0xa174088e ,
    // 0xe4e4daa4 , 0x71cb3604 , 0x87987d00 , 0x1d1d8543 , 0x61fbf37c , 0x72a7bbda , 0xdf30fa82 , 0x48e9ac40 ,
    // 0x67e35fed , 0x023676c0 , 0x3c3af93f , 0xd793b469 , 0xf89d12ad , 0x40c6a78f , 0x28783194 , 0xaadc641c ,
    // 0xc8b163f5 , 0x7af24909 , 0xfbba7a08 , 0x91a76daa , 0x0110920b , 0x2a72596a , 0xe0c7670b , 0xe25bbb03 ,
    // 0x9e130f3f , 0x286de6e2 , 0xa37cc871 , 0xde3f8c00 , 0xcdd00cc2 , 0x965c61aa , 0xfd2d16a3 , 0x52aebd6b ,
    // 0x59865fc0 , 0xfb7d5c5c , 0xd27b57fe , 0x19d345d6 , 0x8c255798 , 0xa62a62ba , 0xec3e7b3b , 0x85b5637d ,
    // 0x214ae6eb , 0x09186b86 , 0x15c513f5 , 0xbc526b75 , 0x137d11b2 , 0x22e2c7f4 , 0xf8c3819f , 0xc34a821e ,
    // 0xcf24075a , 0x242efcf0 , 0x5092b749 , 0xccc08afc , 0x5052bf3d , 0x0b14836b , 0x88ad786d , 0x45a76d4e ,
    // 0x72fdbc14 , 0xe549d211 , 0xe56c3d7e , 0xf5f7d507 , 0xd1face87 , 0x6668d5c1 , 0xbe44a38e , 0x52047163 ,
    // 0x0d6ebcb0 , 0xa2395417 , 0x0fb22d85 , 0xd1194d14 , 0x6dcaf675 , 0x6953eec4 , 0x86313d46 , 0xbb291c7a ,
    // 0x56b57c64 , 0x8d243f64 , 0x2a8f985e , 0x6522af6b , 0xbf6ce5ee , 0x145387f3 , 0xfa7eecf7 , 0xee544df4 ,
    // 0xb1e037a2 , 0xaf11a0d5 , 0xd69d45b5 , 0xed78f57c , 0x854e244a , 0xb4e22708 , 0xc049eae0 , 0x1aa518d5 ,
    // 0xce90c52f , 0x512d47df , 0xeb2e2776 , 0xbd273cba , 0xa6a07e7f , 0x7bf05af2 , 0x01625574 , 0x70c859cf ,
    // 0x6d5a36f8 , 0xa1a2bb8a , 0xf83a483a , 0x2886f2b1 , 0xbf3a56e0 , 0x55df7b20 , 0xe6872031 , 0xace8583e ,
    // 0xef93e505 , 0xc65a7bfe , 0x078e12f2 , 0xd6ed5d2b , 0xcfc1aa0f , 0x5341a30d , 0x37be5beb , 0x7d3d8871 ,
    // 0x4c5f4ffa , 0xa5962c19 , 0xecfda25d , 0xb4c9d71e , 0x0cbcf349 , 0x56f8012b , 0x4f46352c , 0xc673a535 ,
    // 0xa6efa2a2 , 0xe8c5e384 , 0x48f1ee45 , 0x7fd39827 , 0x1c5c101d , 0xd06818d1 , 0xffa5f89a , 0x9353e4ce ,
    // 0x39b5c7f2 , 0x93e6b20c , 0xf32810c2 , 0x410d424a , 0xcf33e3db , 0xf1d93ffc , 0x2371abd2 , 0xe6794e9d ,
    // 0x380db2fa , 0x594a8886 , 0x82367b5d , 0xe6552e97 , 0x3fdc1ea0 , 0xf305357d , 0x7e31068a , 0x5d27cc05 ,
    // 0x32c69f31 , 0x0167f497 , 0x7e6a1761 , 0x00a13229 , 0xb776533a , 0x44949f6e , 0x2a5bb57e , 0xe132d87b ,
    // 0x8ed82aff , 0xac3366f6 , 0x2fc0d961 , 0xdc0ca748 , 0xc963d571 , 0xcd10d6c7 , 0x4af0d83a , 0x22ad425e ,
    // 0x9430330a , 0x57254ab7 , 0xed686c38 , 0xb262df3d , 0xe9b2adcb , 0xb94d5889 , 0xe06d0bd1 , 0x2e480dab ,
    // 0xb0a152cd , 0x367a3f21 , 0xf5cafd56 , 0x9b7c8c0a , 0xba4c3616 , 0xf8e64f08 , 0x60c84233 , 0x9d677ac6 ,
    // 0x9669218f , 0xdbf068e4 , 0xb505eb56 , 0x1fc113a7 , 0x437cd7d8 , 0x1c4df6b0 , 0x5ac7d45d , 0xae804cab ,
    // 0x53874ef5 , 0x3e673518 , 0x9919e141 , 0xf56de048 , 0x06556d48 , 0x30aedadd , 0xaf72a9d4 , 0xb7d8fb15 ,
    // 0xf1c15b6d , 0x33b1c23b , 0x4fc43690 , 0xf5ec12ef , 0x17ef5042 , 0x58cf794f , 0x9f354ee9 , 0xb334e323 ,
    // 0xcc8695f8 , 0x3dd367ba , 0x6026e189 , 0x8bb7cf81 , 0x742db792 , 0xa7a428ba , 0xb281c923 , 0x0c2fefff ,
    // 0x909e7dc2 , 0xef68de3d , 0xc0a63dd8 , 0xff1b2a44 , 0xf235c381 , 0xe166e49e , 0xb38176ba , 0x2872d89e ,
    // 0x418c2090 , 0x5e89e0d9 , 0xaff56150 , 0xe9c32afa , 0x10513bae , 0x2d13faa0 , 0x125511f8 , 0x9b7d2f16 ,
    // 0xf6151b03 , 0x45d4e4ab , 0x75926964 , 0x24a5e019 , 0x9eb8eff2 , 0x51cfb05d , 0xa682eb4b , 0xf7ad94a6 ,
    // 0x7732df95 , 0xbd3dbb29 , 0xd378b2e2 , 0xc85028d2 , 0x73c0d682 , 0x25d61874 , 0x78be145b , 0x4ed37f82 ,
    // 0x4dbfecb5 , 0x14c8148f , 0xd03fd296 , 0xef294e91 , 0x70c44a0f , 0xa430e693 , 0x08a1a573 , 0x414bcbec ,
    // 0x8121e35b , 0x582505c3 , 0x3095abb4 , 0xb71d788e , 0xe0271bf2 , 0xa95ae31b , 0xcdafd26f , 0xf582aff1 ,
    // 0x10896433 , 0x01a23922 , 0xddd42330 , 0xaab93eda , 0xe6f98289 , 0xb642ed7b , 0x1b394182 , 0xdd91cad0 ,
    // 0x3fb9b8e8 , 0x001af9e2 , 0x8d9a92bb , 0x716ac1df , 0xf99489f0 , 0x32c58075 , 0xed14da69 , 0x9f2d1b80 ,
    // 0x45de2ebc , 0x96ab3b86 , 0xddfdf975 , 0xc877a977 , 0x08b526fa , 0x255baafe , 0x6bffc072 , 0xbb7d6390 ,
    // 0x5f25a865 , 0xb82d07b5 , 0xe73dbe9e , 0x8c1705a0 , 0x7e11957e , 0x9b538b04 , 0xf8f46721 , 0xc1bd6be3 ,
    // 0x90f6e138 , 0x365ec1cd , 0x4c0c8ad8 , 0xa9e1b736 , 0xcc77abbb , 0xa43965b4 , 0x5a1e80c7 , 0x3790ebb3 ,
    // 0x80aa1fe3 , 0xdf33b33a , 0x859c42bb , 0x71b6da3c , 0xec36ce2c , 0x27a42009 , 0x49bc6a5b , 0x9f00adbd ,
    // 0x1833ba9a , 0x5c03a1c7 , 0x80f86aa9 , 0xd5413e50 , 0x3439cde0 , 0x2ec57fb0 , 0x7913997d , 0xc2a1f117 ,
    // 0x1813c975 , 0x53694a05 , 0xcfd08e1d , 0xe66f193b , 0x513254ff , 0xf5f6f0b8 , 0xb23c7f98 , 0xbdc309f1 ,
    // 0xaf3cb7c2 , 0x03e469a2 , 0x0961a615 , 0x1a862e43 , 0xb934b11c , 0x5b80247b , 0x206b43f8 , 0x4e0b13de ,
    // 0x272b8f97 , 0xa424fdf8 , 0x4367cc98 , 0xc2f6382a , 0x3c0a52ef , 0xc20827f8 , 0x985c6acb , 0x443d4ead ,
    // 0xf0dd4d6d , 0x16affa44 , 0x63bb3b41 , 0x71311b42 , 0x4b0f34d3 , 0xa389031d , 0x9d024dcf , 0x9362a49d ,
    // 0x7bd2513e , 0x218a8f74 , 0xe9daac00 , 0xae9a9a40 , 0x05e830eb , 0x115fb72f , 0xed69c4da , 0x0193c285 ,
    // 0xdab06bed , 0x4a890ba0 , 0xede875c0 , 0x4606488a , 0x50157af1 , 0x4fd5a859 , 0xca7f36ce , 0x85a371c7 ,
    // 0x91798f14 , 0x58c0b61b , 0xb5c2d824 , 0x94803b9a , 0xb91bb396 , 0xc55fabf2 , 0x6f18c384 , 0x01bccb78 ,
    // 0x1a533bb3 , 0x130633ef , 0x91d1424e , 0x84f8788b , 0xb9cdd05b , 0x1b7e4755 , 0xb687c92e , 0x1fbf7fd0 ,
    // 0xe1790294 , 0xdbdc9182 , 0x4e5bc139 , 0x8a7237a3 , 0xfcdbd22a , 0x818add97 , 0xf7654b2b , 0xe537c88c ,
    // 0x9dce8147 , 0x8ff1faf0 , 0x7dbc4e95 , 0x23e06c41 , 0xcb9855e6 , 0xea861e33 , 0x60c6907a , 0xb9d395c6 ,
    // 0xfe109ca3 , 0x365be0e6 , 0x35cba7e3 , 0xf6f88bf9 , 0xc5404014 , 0x937bfe73 , 0x09423d61 , 0x9fc35ae8 ,
    // 0x2bfdc9d4 , 0x115405b4 , 0xfd33d922 , 0xca20a60e , 0x76556bc0 , 0x1eb1a222 , 0x3c6692ef , 0xe2e93601 ,
    // 0xbc080f9b , 0xa5843c62 , 0xf53031ce , 0x2d7e12bf , 0x1a0e3dd7 , 0x3e0d750c , 0xe8b9a962 , 0x8a6c1fcd ,
    // 0x7a685989 , 0x29209c90 , 0x9dbc3f79 , 0x0e3c7558 , 0x38695110 , 0x1c97111e , 0xad2e2bfd , 0x194ac45c ,
    // 0xce869454 , 0x1e154de8 , 0x1700db54 , 0x6fa31781 , 0xb55a5f13 , 0xa2adecbe , 0xe4709c72 , 0x748d00b8 ,
    // 0xdabe3974 , 0xb3a81a52 , 0x2f7683e4 , 0xb6e503c5 , 0x5674d357 , 0x43706b96 , 0xe2ceaf61 , 0x40fa5e8a ,
    // 0xaa733f96 , 0x7266bdaa , 0x526c6988 , 0xe3174696 , 0x3f10f22b , 0x6f5f33b1 , 0x0d8ebcf9 , 0x09554c0f ,
    // 0xe2499364 , 0x2cd0e56d , 0xe4748882 , 0xe30badc2 , 0x9f4541d2 , 0xfee81859 , 0xf7dabd80 , 0x64cc30b7 ,
    // 0xf1b97277 , 0x98911a43 , 0x822f651c , 0x116f61d6 , 0x2fba7e44 , 0x0ca56638 , 0x0a1398f6 , 0xdf49a111 ,
    // 0xbb1e3d72 , 0xfe269c2e , 0xbf40b795 , 0x7692c6c9 , 0x7fee9bc0 , 0xa449869a , 0xfef9e8a4 , 0xb52abc3b ,
    // 0xd6b1df23 , 0x07bc0b43 , 0x7cae9ca7 , 0xa1fa2510 , 0xed25edc4 , 0x719db801 , 0x3f92f8d1 , 0x84f7ff7a ,
    // 0xd34619b2 , 0x07809643 , 0x91238e94 , 0x3bb13b80 , 0x02a78c9b , 0xc672546a , 0x5e2d7405 , 0x85fdc0f1 ,
    // 0x0585438e , 0x7925ea85 , 0x04f79a64 , 0xe6d7f86b , 0xb3cf44a7 , 0xc451a05a , 0x9e4c10fd , 0x1828836e ,
    // 0x02325e02 , 0x2e3981d9 , 0xe2057a1c , 0x73f40aa1 , 0xd8b49a2e , 0x1d4b60cd , 0x846ee8b3 , 0x58e0efea ,
    // 0x90dfd093 , 0xb9457b66 , 0x4b3409bc , 0x140fc279 , 0xd0ebf34d , 0xc063a952 , 0xecd57eae , 0x70e5bda4 ,
    // 0xcb2d80ba , 0x2bc5383f , 0xf80debe3 , 0x88e614df , 0xf729eb40 , 0xf792b83c , 0x8e2796b4 , 0x33e4b6eb ,
    // 0xbfcf1784 , 0x2439dc92 , 0x3ebabbbd , 0x79143883 , 0x452321e0 , 0x495a1e4b , 0x5dd7876a , 0x00000001
  }
};
```

C.7 Mersenne Twister mt64 RNG class rng_jump_mt64.h

```
// -*-c++-*-
// rng_jump_mt64.h
// Mersenne Twister Random Number Generator
// with statistically independent instances
#include <cstring>
#include <bitset>

class rng {
  static const unsigned N = 312;
  struct state {                // state information
    unsigned n;
    unsigned long long y;
    unsigned long long z[N];
  } ;
public:
  // the first instance is initialized from the seed
  // subsequent instances are jumped ahead giving independent rngs
  rng( );                       // constructor (called with no arguments)
  rng( const unsigned long long& seed ); // constructor (called with seed argument)
  unsigned long long gen( );            // rng returns a 32-bit unsigned integer
  static void setseed( const unsigned long long& seed ); // call once for all rngs
  static const int wid = 16;    // integer format width
private:
  state s ;                     // rng state
  unsigned id ;                 // rng sequential id per instance
  void init( );                 // initialize a rng
  void seedgen( );              // seed the first rng
  void jump( );                 // jump to new state
  void jpunpack( const unsigned long long* const Jump_P );
  static unsigned long long seed; // seed for first generator
  static unsigned ref_counter;  // number of instances
  static state ref_state;       // zero state of newest instance
  static unsigned long long Jump_P[][N]; // jump polynomial coefficients
  static const unsigned int NN = 312; // rng parameters
  static const unsigned int MM = 156;
  static const unsigned long long int MATRIX_A = 0xB5026F5AA96619E9ull;
  static const unsigned long long int UM       = 0xFFFFFFFF80000000ull;
  static const unsigned long long int LM       = 0x000000007FFFFFFFull;
  static const unsigned CPdeg = 19937; // characteristic polynomial degree
  static const unsigned Jdmax = CPdeg-1; // max jump polynomial degree
  static bitset<CPdeg> JP;      // jump polynomial
};

rng::rng( ) {                   // constructor (called with no arguments)
  init( );
}

rng::rng( const unsigned long long& seed ) { // constructor (called with seed argument)
  setseed( seed );
  init( );
}

void rng::setseed( const unsigned long long& seed ) {
  if ( ref_counter == 0 )
    rng::seed = seed;
}

unsigned long long rng::gen( ) {                // rng returns a 32-bit unsigned integer
  unsigned long long int y;
  static unsigned long long int mag01[2]={0x0ull, MATRIX_A};
  if (s.n >= NN)             // generate N words at one time
    {
      int kk;
      for ( kk = 0 ; kk < NN - MM ; kk++ ) {
        s.y = ( s.z[kk] & UM ) | ( s.z[kk+1] & LM );
        s.z[kk] = s.z[kk+MM] ^ (s.y >> 1) ^ mag01[(int) (s.y & 0x1ull)];
      }
      for ( ; kk < NN - 1 ; kk++ ) {
        s.y = ( s.z[kk] & UM ) | ( s.z[kk+1] & LM );
        s.z[kk] = s.z[kk+(MM-NN)] ^ (s.y >> 1) ^ mag01[(int) (s.y & 0x1ull)];
      }
      s.y = ( s.z[NN-1] & UM ) | ( s.z[0] & LM );
      s.z[NN-1] = s.z[MM-1] ^ (s.y >> 1) ^ mag01[ (int) (s.y & 0x1ull)];
      s.n = 0;
    }
  y = s.y = s.z[s.n++];
  // Tempering
  y ^= (y >> 29) & 0x5555555555555555ull;
  y ^= (y << 17) & 0x71d67fffeda60000ull;
  y ^= (y << 37) & 0xfff7eee000000000ull;
  y ^= (y >> 43) ;
  return y;
}


void rng::init( ) {
  // the first rng state s is seeded as indicated
  // for subsequent rngs compute state s
  // jumped ahead from the reference state
  // set reference state for next jump to zero state of current rng
  if ( ref_counter == 0 ) {
    seedgen( );
    jpunpack( Jump_P[0] );
  }
  else
    jump( );
  ref_state = s ;
  id = ref_counter++;
}

void rng::seedgen( ) {
  // seed the system
  // set seed for first generator (algorithm specific to rng)
  // the rng must be in state zero, not the seed state
  s.z[0]= seed ;
  for ( s.n = 1 ; s.n < NN ; s.n++)
    s.z[s.n] = (6364136223846793005ull * (s.z[s.n-1] ^ (s.z[s.n-1] >> 62)) + s.n);
  gen();
}

void rng::jpunpack( const unsigned long long* const Jump_P ) {
  for ( int j=N-1; j; j-- ) {
    ( JP |= Jump_P[j] >> 32 & 0xffffffff ) <<= 32;
    ( JP |= Jump_P[j]       & 0xffffffff ) <<= 32;
  }
  ( JP |= Jump_P[0] >> 32 & 0xffffffff ) <<= 32;
  JP |= Jump_P[0]       & 0xffffffff;
}

void rng::jump( ) {
  // compute jump state t.z[0 .. (N-1)]
  // from jump polynomial JP and base states sy=s.y
  //
  // t.z[0  ] = J[0]*sy[0  ] + .. + J[d]*sy[d    ] + .. + J[Jdmax]*sy[Jdmax    ]
  // t.z[i  ] = J[0]*sy[i  ] + .. + J[d]*sy[d+i  ] + .. + J[Jdmax]*sy[Jdmax+i  ]
  // t.z[N-1] = J[0]*sy[N-1] + .. + J[d]*sy[d+N-1] + .. + J[Jdmax]*sy[Jdmax+N-1]
  state t;
  unsigned long long sy[Jdmax+N]; // Jdmax+N states 0 .. (Jdmax+N-1) for jump
  s = ref_state;                // start s at ref state
  for ( int i=0 ; i<Jdmax+N ; i++ ) {
    sy[i] = s.y ;               // save states sy[i in 0 .. (Jdmax+N-1)]
    gen() ;
  }
  memset( t.z , 0 , sizeof(t.z) );
  int i, k;
  for ( int d=0 ; d<=Jdmax ; d++ ) // JP term power = d in 0 .. Jdmax
    if ( JP[d] == 1 )
      // i in 0 .. (N-1)
      // k in d .. (d+N-1) for each d
      // k in 0 .. (Jdmax+N-1) overall
      for ( i=0 , k=d ; i<N ; i++ , k++ )
        t.z[i] ^= sy[k] ;       // k = d + i
  s = t ;
  s.y = s.z[0];
  s.n = 1;
}

// number of rngs instantiated
unsigned rng::ref_counter=0;

// reference state for jump computation is the
// zero state of the previous instance
rng::state rng::ref_state;

// seed for first rng instance
unsigned long long rng::seed=0x12340f00;

// Jump polynomial
bitset<rng::CPdeg> rng::JP = 0;

// MT32 jump coefficients
unsigned long long rng::Jump_P[1][N] = { // jump polynomial
  {
```

```
// 2^20
0x9e48e623e6ef89acull , 0x87b1a56b296c3ef9ull , 0x10aeda6af61caae0ull , 0xa16a4bde28d92e76ull ,
0xadfa625443e8e61aull , 0xe38e7067c617c885ull , 0x166253e3322f6158ull , 0x5056c554950b31bfull ,
0x130eb6e39d7ad87cull , 0x7f4c407b96762a9eull , 0xbc8c98018ca42703ull , 0x42212770d8d49545ull ,
0xbf5e329e09895a6eull , 0x0532b55cf3a2b230ull , 0xef0d72075fc84014ull , 0x962280b6350d53e2ull ,
0x584d4e621b895f72ull , 0xf536b2d997b48146ull , 0xa0db8ed98ef0b09cull , 0xbb0b0747c308d595ull ,
0x8ff483831480a1d0ull , 0xeefa395aec7c0e7eull , 0x59b1369f7e3fdd00ull , 0xbe8b8269219485c2ull ,
0xf10ff0e88b4581bfull , 0xf10b24af748569b6ull , 0x6c553116080f3816ull , 0x12d7275a3a5eb5c2ull ,
0xa08349e876c20596ull , 0x71a36f46b8f6548eull , 0x0eb668b76d50fbf9ull , 0x1807ebc5cb0c51deull ,
0xcebf0fc45b51bafcull , 0xf4f905544545d811ull , 0x3aad0cc19fb3bcc5ull , 0xe4c259df5e33e219ull ,
0xa6ccf12509a68757ull , 0x657f4c17d333c450ull , 0xb76d25a631eaedbeull , 0x08956ca47ad62df9ull ,
0x0318393e711515f6ull , 0x0568088c9625c952ull , 0x8e0a3d31c919626bull , 0xfbe9b9e0b8a69d34ull ,
0xef945eeb16eb2e62ull , 0x357245aabb65e408ull , 0x929c958add7e6814ull , 0x1af3c4b01d673dcfull ,
0x70ff1940d526ec55ull , 0x299d537923091a6aull , 0x2f24a8abd25ab4fcull , 0x14b00ba9ef2458fdull ,
0xf0a4195abf739b18ull , 0x0b856806a512b24bull , 0xcaa9f682000173cbull , 0x67feadefc34016f3ull ,
0x63f659f499d67d18ull , 0x4b05af1763513442ull , 0xd8f3717e52bd9020ull , 0x15ba53896db7f259ull ,
0xf7e392386475cdadull , 0x3b3882f8997c4340ull , 0xfa4edf4ac31177d7ull , 0x6c0786ef00c7aaddull ,
0x1932a35ee22974bcull , 0xe085992912c6be0cull , 0xd1013369720f66a2ull , 0xb457d34fa080f275ull ,
0x41448350a2b10546ull , 0xb7697600b5107d0bull , 0xd5eaec5cc3eb1b6cull , 0x6c17a3decd0ca3eaull ,
0x6cf0e8a450ad8d2dull , 0x697f9b384f152ca5ull , 0xae75f9d8d3738734ull , 0x85a24675e7a6aa69ull ,
0xc58e61afd44aeb3dull , 0x87d739de4723e6c8ull , 0xbb62acfe53ee38b5ull , 0x3025709532e5296cull ,
0x11490883cd1846dcull , 0x5956dac229009f7full , 0x28066ebd8a7a38bcull , 0x2d897c6bf19b5c5bull ,
0x87229a26d0dbbb61ull , 0xe3fa47a930b4b844ull , 0xdc49472496444e6aull , 0xfc76ff3ee66ed123ull ,
0x3b98e618ac9463e3ull , 0x6918af039820a7dcull , 0x32a21e01b32f796bull , 0xf4fafa74dc2b7884ull ,
0x91712e41dc0c888eull , 0x513edb0bdb83fe50ull , 0x52a85df1389779e9ull , 0x174604f4b488d9deull ,
0x627ef0cc8f152883ull , 0xa84559473127a429ull , 0x4d8b9e7552d79342ull , 0x25a327ec1daa9eeaull ,
0x0d9a8dd252775bbaull , 0x533eea945385f59cull , 0x2704be952e3549e9ull , 0xe37674ed39b1cdc9ull ,
0x2fc545b16a57a14cull , 0x63fc61abaf68d4b4ull , 0xd915de012a6e5b2dull , 0x9cc77871ef19f234ull ,
0x5fbb11c5319928beull , 0xd95aeb6f03d2a52cull , 0x00fd99f0fb924d07ull , 0x8a6bc574a440a366ull ,
0x9a9a8d54a0662ac9ull , 0x05d0b3538a6f9910ull , 0x973c4d2ce0fad24cull , 0xfc39ca9568993d80ull ,
0xfc70e7a657deb613ull , 0xa0aaaddbf48c6e68ull , 0x77429d5203eb965full , 0xd54e695b4771689full ,
0xb66d068a69d9de0dull , 0xb415a66a8c66aea9ull , 0x2aa41108f2a30de1ull , 0x1768cbad7643d4d2ull ,
0x7953e63abf30891full , 0x7ba41fd0ed19e60bull , 0x6796637e24f7e0a2ull , 0xfa234881349afb7cull ,
0x5eb8b7a5d128c130ull , 0xf8ee4128529c66a1ull , 0x6c84f9fe3874e779ull , 0xe19a59ee9f4440a5ull ,
0x9218121f793cba64ull , 0x679fde4577496180ull , 0xbd3fe66ae73f297dull , 0xca18bb4a6ef67545ull ,
0x1e28572dd7a08888ull , 0xc20b7d76eddda2d0ull , 0xa53bf2647d4befbbull , 0x2ec61ee38b1cb88aull ,
0x5662faa613f3e082ull , 0x93bedbcd73c29204ull , 0x40bc424d5f79d5eaull , 0xa367ced03f8f0270ull ,
0xe4dab089dc6b857aull , 0x25b289e334046b2full , 0x94771faa0016d93dull , 0xb4f76bd83a673955ull ,
0xea75dbe3ad33bacbull , 0xb63a741eac495162ull , 0xcd40da70558b4a48ull , 0x38e84f50f4cadd2cull ,
0xe1335a5424d33ff1ull , 0xe4a63af6700672d3ull , 0x582a6a393d66b326ull , 0x05cbddb22eddd4d3ull ,
0x570ae279e6f12517ull , 0xe87e04bcb3de0eb0ull , 0xb362acd897dd8b8aull , 0x547d063de839f509ull ,
0x54cb31b00a305758ull , 0xc938192e9f93cd8eull , 0xe0c5488c2dcebdc4ull , 0x7e1238facc7356d7ull ,
0x9c7618a83887bf8eull , 0xa0d58650032e0723ull , 0xf9c8480f25b5049cull , 0xb1e01da354e632a1ull ,
0x358529c6d991f5cfull , 0x28bc59474462d4cbull , 0x5b76e9dd51cc2ad3ull , 0x5b7fd1381b766c23ull ,
0xddf46ff1730dad06ull , 0xc52a6d156bde1b8eull , 0x4c8334e796e27cd4ull , 0x6ab3b24074552029ull ,
0x160203dee6d9511cull , 0x11cd1426e0a24587ull , 0xa814dd8dc4bd363dull , 0xe099e812018b80e4ull ,
0x7c6cf7f91bd33c0eull , 0x3c4c091f56a8db24ull , 0x3250825008b1ec1aull , 0x7e833aacdcc1455cull ,
0xd456f9beac0d563cull , 0xa83e860446988702ull , 0x7df60fb512286c11ull , 0x5bdfd4297a37965aull ,
0x7dc9dd8699aac3e5ull , 0xcd89e3eb4fe6ac27ull , 0x29e39e5138353bf6ull , 0x230f167dc8ea060bull ,
0x1577a1c6c47c58a0ull , 0x79a227af3cc2fc00ull , 0x639e7d13892c943cull , 0x787e77f8ac8ee42bull ,
0xac6cf9b6f9579eeaull , 0xa488b46968731aefull , 0x139a94aade0f47deull , 0x814e621de9e6c2b9ull ,
0xa10d0f10dabcf9c9ull , 0xfd8a036add2b4fbcull , 0x3d0a0fb46a18f587ull , 0xfd3413e3c3a32bcbull ,
0x723c7c52c8753946ull , 0xce2b4a8d22b1a950ull , 0x19ea7f75fdd4f83dull , 0x6cfd9623b74ecf92ull ,
0xde287f40e4caba52ull , 0xe53636528baca150ull , 0x7a2c656a0d388e20ull , 0xcd43e904a2026126ull ,
0x1e115b4df3bf5befull , 0x1cbd010558f3bdadull , 0x4bf4ea4936acacc7ull , 0x892f1cfe318192dbull ,
0xb505b9eb8bb9ecbfull , 0xecdfa31d299f3bf2ull , 0xa6ab99174f46e708ull , 0xe61d015825118640ull ,
0x741da69a6d96a4c7ull , 0x2de7a919496bccdaull , 0xd7cc3d034c8c0149ull , 0xe750a0b4a6580f2eull ,
0x458857d6c9c0ee6cull , 0xb9648ad6e1741198ull , 0x919b9fb950081152ull , 0x130d58687b84db2full ,
0x198bb76e7d155a28ull , 0x2e69a3c177a94c0eull , 0xd695d8c6af736e3full , 0x6bcd3d9d81eaabc7ull ,
0x7191cb2138ddede5ull , 0x9732f809b35b8a91ull , 0x4fe98c4ee0a32c59ull , 0x4c58d026a0e30e27ull ,
0x613d63ca59e57a99ull , 0xba72d55b1a3bc20full , 0xbae8ba9557ae231bull , 0xb2cc7386cf967d26ull ,
0x48f9c729828acdbeull , 0xc92988357e3270f3ull , 0xb13c2c0e84548909ull , 0x65ebcdd5392d4d46ull ,
0x2a6bd22a5afce3c8ull , 0x2205f1fc4b2d99a5ull , 0x7ce884258b403058ull , 0x4559fae2c9573f01ull ,
0xd29b67d8245dd4bbull , 0xf9dc1375da31ecaeull , 0xa7852757a1875822ull , 0xf4a06ae488fa04faull ,
0xb52d848e88c5fe99ull , 0xdfa3045e80aa1e00ull , 0xf66c57b57777a19cull , 0xe6348bf7dcc7cdc4ull ,
0xa3909f3741c39437ull , 0xe9070656c15fc921ull , 0x5844c9261966d0e1ull , 0x9654291702475d4cull ,
0xd61b4b8067f04f70ull , 0xae7f9f05c04f9a12ull , 0x10d39ffed3bb7196ull , 0xc4a37af13746625aull ,
0x4454ac1e5325d3f0ull , 0x7ba91f277da0ba04ull , 0x5051bbdf79e40c8cull , 0x725e3d79b0ce59f2ull ,
0xf952ee4cbb239291ull , 0x8135d1300c82b33eull , 0xa234a644e8d7715full , 0xcb1b2b0048fce64eull ,
0xb636ff6ba69e4b0bull , 0x8a492e68a94e24f5ull , 0x64a7db5a789e4687ull , 0xf597e6d649379b9full ,
0xea681fafadea8178ull , 0xe56cc7029be30279ull , 0x776f099af215d2c0ull , 0xd06c0ec8ad7dc968ull ,
0x48f5b4f755d06e32ull , 0x660392fbb0392cccull , 0xef5cbf47a24b7836ull , 0x4ce3dc55f60ee4b5ull ,
0x360d2fe075bdb14full , 0xc78e3d1080895ac7ull , 0x64bd58337b530242ull , 0xf75fe8f3cf1ceb0full ,
0xf7fd81ebe86232c0ull , 0xa34cc64076c136a7ull , 0x74311c1440a7b6f8ull , 0x99407b17a582d52eull ,
0x5a65f6e0bd5da29aull , 0xaddf5c2fcc9452c2ull , 0xf80f2281b0cff619ull , 0xe6a58d7109e7e770ull ,
0x4c64028a7321da10ull , 0xcaa460c505d236f6ull , 0xf85249abef239557ull , 0x639ffded4c8c0b6full ,
0x22136dba420d6b6dull , 0x998a57a0b7bbfffeull , 0xd173253d4792cb0aull , 0xe386f010ea23c96cull ,
0xed550e865adf2ea8ull , 0xd0cd7f485dd7f626ull , 0xdfdc2f073bad8763ull , 0x535372ddd1cd0d15ull ,
0x4112af74a6250027ull , 0x02398e6abfbe9330ull , 0x99d20041b0736539ull , 0x0000000044337dd9ull
```

```
    // 2^10000
    // 0xe4115b36993bffe7ull , 0x2290d948e83a43acull , 0x2277e12bc9c074b4ull , 0x33c19b90e1b9deadull ,
    // 0xfff75a273cd88961ull , 0x676703d88a3bf981ull , 0x1e833b82c93e2f0bull , 0x4f09b86e86159fd9ull ,
    // 0xf7fc813dd4bae594ull , 0x544d0855917c2fd1ull , 0x7c46429c52e0d4adull , 0x7b82ef96b5e8b90aull ,
    // 0xea3b8b69d445696bull , 0x391aeaffcad402e3ull , 0x37fc607c3c5f0e96ull , 0x4e6ff83ca4ffe513ull ,
    // 0xf54d97e0ebe18662ull , 0x63869eee5f596911ull , 0xb8e381d1782841abull , 0xad664404a638f01dull ,
    // 0x6cb3b4ee2f7e0de5ull , 0xf309adbdd4220ccdull , 0x7e620e6e7660b595ull , 0x659dea15686f6aaeull ,
    // 0xbe19ae26ca4f5300ull , 0x079baed9cd0906a4ull , 0xd2736da73c95794full , 0x91eab4989d8f1ce2ull ,
    // 0x10f70a76dbc6ad2eull , 0x5a51b212b9a0831bull , 0x7c14e6393859c03aull , 0xe7ff466509194169ull ,
    // 0x9fd856495627c455ull , 0x0cecae753e7420adull , 0x15549342017d2da0ull , 0x61e3fca166da2712ull ,
    // 0x85115a2a4452a8f6ull , 0xfc9bc316f3b025e4ull , 0x8735faa0107a7eadull , 0xe48d5396b2ef2a90ull ,
    // 0xd5331a0850bf8980ull , 0x193ce2085edc6c0eull , 0xc8927fd78ff13d62ull , 0x192826460ae8b0c5ull ,
    // 0x515a702d83b7658dull , 0xc66bc86d14f1d331ull , 0x910bea7b497abaf4ull , 0x213b44512d139a17ull ,
    // 0x73dde418b88e53a5ull , 0x052ba71c4c726681ull , 0x095ee33eddc88352ull , 0x45bdb5ea5a80a229ull ,
    // 0xd27f26ada47f37f0ull , 0x8093e3f5eafcb507ull , 0x2b37a0de97125d0bull , 0xb0f2139f78dd50ddull ,
    // 0x09e17648d0706b0aull , 0x4e27106995e7cd98ull , 0x6e26f66817ef329dull , 0x4aa4ad7e442865b2ull ,
    // 0x52fd314c50a6b3e7ull , 0xe8456586297599edull , 0x22a1943afe2e2761ull , 0x77e3027018b859b7ull ,
    // 0x86719cebc0b84b6cull , 0xb5f0693a9d64652bull , 0xfb34e899be436b5aull , 0x3297798786df0b35ull ,
    // 0xa05b817205507b5cull , 0x5ef055acd7fa0c82ull , 0x3c66cd88c309d684ull , 0xe809f6c5e0cf2555ull ,
    // 0x0e4a7d1635e6e11dull , 0xf9ab85ae5da1c0c6ull , 0x835bbebf54cf56ffull , 0x43763a856ea0a9b4ull ,
    // 0xff2a0f8420af8ef4ull , 0x7ddd80b1a003cc14ull , 0x935ed3046eb069e0ull , 0x1ae06506dd33c79full ,
    // 0xe0b57515e8bc958aull , 0x4eb2eb1a0793912eull , 0x2cbd8ce05a2cc449ull , 0x48f7bbdbdb04ed8aull ,
    // 0x25cd43bc7b57c203ull , 0x90b7e7224988918bull , 0x719cab883b5c17d3ull , 0xd267ddd5d0f1e45full ,
    // 0xd83b034ed5004c98ull , 0x5b641cf2fbf9ebf7ull , 0x4b500a195c138beaull , 0xf0e775a546bb9b8cull ,
    // 0x47bc91a0515eb470ull , 0xcb878cc53fad6c93ull , 0x9a6474c53da37aa3ull , 0x168b8f4410c77948ull ,
    // 0xd861af0c01bdb29eull , 0x6c8c890b5fdc9fccull , 0xc1e9e90123dccdb8ull , 0xfe01089db698de64ull ,
    // 0x1bf4733e638071baull , 0xb7011ffb6f4435c9ull , 0x4fad07c1550bdf20ull , 0x239decb2d26fede2ull ,
    // 0xe67e17c3338c92c8ull , 0xc1a0ee30795c6d75ull , 0x1b410e29808de987ull , 0x8c62bab3b927dd5full ,
    // 0x07036355bd4d79e5ull , 0x38d467b9274591e6ull , 0x711fe3bc4a56c662ull , 0x727959d6f65b3a1full ,
    // 0x792f81f225e915dcull , 0x12ef3163247d7379ull , 0xbd5fb88bad0ba330ull , 0xb7e7f0436108c4dbull ,
    // 0xfbc07978eb393045ull , 0x6d1441a3f9851dc3ull , 0x22d599b1da7b9ee5ull , 0xef0b40f08c1b9438ull ,
    // 0x004b91ab18c2a9efull , 0x557876d5e4d9ac7dull , 0x4831fc302261acaeull , 0xb5dad81bd234b39eull ,
    // 0x7682023d2a49464eull , 0xdfb0b41fd3982e30ull , 0x67f8325a6a30152cull , 0x6113d49dba637bdeull ,
    // 0x44db4e0532d8510bull , 0x717b47fbf70d5c5full , 0xd9d452fe19690c77ull , 0x4d2849a4c44a4c81ull ,
    // 0x50f821f4615a7fd5ull , 0x37486ca8e8d46bcdull , 0xc4a9ffc7e3f525f5ull , 0xd8a0c62f35f1a934ull ,
    // 0x990535d74126aebfull , 0x63d07c17fa2a4250ull , 0xf348076ea0d48ba5ull , 0x930045349d782c3dull ,
    // 0xc8be33a8fffd02f2ull , 0x3e293178ea7ffc8dull , 0xe3cfef2f7cee9c90ull , 0x1c3ad0005a53d5caull ,
    // 0xff32fc0bc88c36b5ull , 0xaae5baabb8af4045ull , 0x5882b7cc6a1baaa9ull , 0x217d4dc96a41f311ull ,
    // 0x43b8bc88be3e1e0eull , 0x41ef00cfff38090eull , 0xd5819f4b24a896a0ull , 0xd8ee01dfaf4845c6ull ,
    // 0x3cf391ac21ceded5ull , 0xe45dfcdc59020adaull , 0xee442aeb69b6b185ull , 0x514aab202a0414a1ull ,
    // 0x94f9c8bbd82b12feull , 0xf8dd6e0e3e82cda6ull , 0xcee3e0e6fda76014ull , 0xee0ccf686463c093ull ,
    // 0x30239324f066d2baull , 0x489b5032e4ae763aull , 0x514bddab718ee7ecull , 0x227385bdb107c84bull ,
    // 0x3740faf2e0ecaa35ull , 0x3505a46928686b95ull , 0x1c8eb2168a63c679ull , 0x9f624dab15c50a70ull ,
    // 0xa25522abbb4e6503ull , 0x68e583ddffdabe4eull , 0xf41f2ac3f1222373ull , 0xac80782bddcafc2dull ,
    // 0x40d89ab57e0608cfull , 0x345351fc2130979cull , 0x372641f5601f65f6ull , 0xa746c0d10b51e6e5ull ,
    // 0xbda6be36455bfb62ull , 0x2c8f875cc9f51502ull , 0xdaa825db0dad07a1ull , 0x6dc7e7cb53616925ull ,
    // 0x1e595b93e437a56eull , 0x215bba08bea01d8eull , 0xf5570592d31532ccull , 0x3beb9b5b1a0bda28ull ,
    // 0xb65f114ae2c72774ull , 0x919889fd8b1ace2aull , 0x01cb2b8de0c2e998ull , 0xf7091c58a42c4229ull ,
    // 0x111a3502ac2391a1ull , 0xd156382ef071f675ull , 0x3714ba5d26a175e3ull , 0xf8f04fad0c014891ull ,
    // 0xb7c4063879db7cf7ull , 0xab30125457df2e67ull , 0x7fd44d82c8ca2145ull , 0x25147530a70ccc4full ,
    // 0x8adc73fd6d17ba1bull , 0x27ac93aaa8ab83feull , 0xa5c0908af55ea944ull , 0x41f5c08cdb6997e6ull ,
    // 0x30ddf2fefa1f1beeull , 0x91eff0dee14f8c6cull , 0x0b5bbf68f6bf8e8dull , 0x533ba3687e45cbc7ull ,
    // 0xfb907e989718fdd3ull , 0xea3315972411225eull , 0xa0f26aba7293d313ull , 0xfbafb565290dc758ull ,
    // 0x559d50a0c9cc35d1ull , 0xb1b4bcc4c1012f5dull , 0xf6a64a4b668159d6ull , 0x8295deea6cb8dc35ull ,
    // 0x6659fe605067eab4ull , 0x6900237741fc96beull , 0x0237306c4cfe7ac7ull , 0x94784af94e051c1cull ,
    // 0x35f2ab22d3e07ebfull , 0x30d0f32872a8eba4ull , 0x77c276bae36b9fe6ull , 0xbb326353fcba584cull ,
    // 0x89a04e58ffaa1242ull , 0xa0907c638e73ec62ull , 0x4101d4491827e5baull , 0xb3d40bc07c91ce8full ,
    // 0x8130235729aaea77ull , 0x26361a872163012cull , 0x85595a36b06da11full , 0x6d5cd34a95fb1355ull ,
    // 0x990c11b19bd74bcaull , 0x2fb2b6a748cc74d7ull , 0xe8592fdcc4cfd658ull , 0xa32720740821de5dull ,
    // 0x4772ebd589571283ull , 0x89094e3253332726ull , 0x5638837333f3fb98ull , 0x018b307dc21da55eull ,
    // 0x35af01d1793eabb6ull , 0xce831706217b7abcull , 0xd76192e60b0db90dull , 0xdbec797e14385e71ull ,
    // 0xfe784e81a58b4684ull , 0xdb7215a1fdf38984ull , 0x19ffed6da1dd16e1ull , 0xe979b10a385e75b1ull ,
    // 0xe153193e61d62943ull , 0x4e986c7b876264a8ull , 0x569b1b2e61c2bb66ull , 0x72f981e908fa0dc3ull ,
    // 0xdf57ba0b02278bb0ull , 0x84bf35d11d58096cull , 0x39cab41e461ee546ull , 0x9a5a8f69ad1a2b3bull ,
    // 0xe4446755d0acd36aull , 0xd23f9baaff0302a0ull , 0x05294873c051cd53ull , 0x1c607c62af6767b9ull ,
    // 0xb6dfb3226f22d513ull , 0x1fce3748609a4889ull , 0x9ad0a97ee0611a6full , 0xab438f8e6825b4baull ,
    // 0x31fc6240e3182d8cull , 0xecfc9d35c95a953dull , 0xcc51c0865c76dc3aull , 0xae8582bddcb407abull ,
    // 0x8724aa12a5b781deull , 0x4dd48855cb8390f7ull , 0x0e42b7f914de69bdull , 0x3993801ad83fe067ull ,
    // 0x3a54e800d7e42e8bull , 0x5d8d44e92f3fbec8ull , 0xc800bb32aabce373ull , 0xcc90dfcafa49f98full ,
    // 0x62533fc07122085dull , 0x235b72618314d2e6ull , 0xe2b74c331b4de0deull , 0x0b41b7abba3a4cdaull ,
    // 0x6e1ae210b67a4499ull , 0x2dd60c417df6e2a6ull , 0x30c0808d0ee406baull , 0x824d60d26c0d3375ull ,
    // 0xfb234c25afcc00b7ull , 0xffc6b5c3641aff53ull , 0x8e5b31ef54055d29ull , 0xd4197110e5e70bb3ull ,
    // 0x624856bec16ec601ull , 0x787a11775bf0645eull , 0x39bf99590959ed5bull , 0xebab880bb8be0182ull ,
    // 0x1f3d9de67c3f8354ull , 0x187adaae34d97b1cull , 0xb8f5dd4ec234a350ull , 0xda61e54cdcdf22f9ull ,
    // 0x6b79ee94dd67be2bull , 0xe8e37f77f519faceull , 0x20539659c39253f0ull , 0x0f23f25a0eab6773ull ,
    // 0xceb110b52b78449bull , 0xf7f488c5911c538dull , 0x92afae9e0c7c01fcull , 0x77cc28e3468db290ull ,
    // 0xf05895253177fec4ull , 0x7fd533f6bd57f96cull , 0xc58fc1bd74828485ull , 0xd71e770f00ae0746ull ,
    // 0x2721fdb614a2831full , 0x2e0da59be7433f6bull , 0x2a099cc7588a726aull , 0x30d38b36950c3000ull ,
    // 0x26bff7143c5d680eull , 0xa3cff807e157c93eull , 0xbe88f3d6acb68f48ull , 0x00000001b3fdecd4ull
  }
};
```

NO. OF COPIES	ORGANIZATION
1 (PDF only)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA 8725 JOHN J KINGMAN RD STE 0944 FORT BELVOIR VA 22060-6218
1	US ARMY RSRCH DEV & ENGRG CMD SYSTEMS OF SYSTEMS INTEGRATION AMSRD SS T 6000 6TH ST STE 100 FORT BELVOIR VA 22060-5608
1	DIRECTOR US ARMY RESEARCH LAB IMNE ALC IMS 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRD ARL CI OK TL 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRD ARL CI OK T 2800 POWDER MILL RD ADELPHI MD 20783-1197
	ABERDEEN PROVING GROUND
1	DIR USARL AMSRD ARL CI OK TP (BLDG 4600)

NO. OF COPIES	ORGANIZATION
1 (CD only)	ASST SECY ARMY ACQSTN LOGISTICS & TECH SAAL ZP RM 2E661 103 ARMY PENTAGON WASHINGTON DC 20310-0103
1 (CD only)	ASST SECY ARMY ACQSTN LOGISTICS & TECH SAAL ZS RM 3E448 103 ARMY PENTAGON WASHINGTON DC 20310-0103
1 (CD only)	DIRECTOR FORCE DEV DAPR FDZ RM 3A522 460 ARMY PENTAGON WASHINGTON DC 20310-0460
1	US ARMY TRADOC ANL CTR ATRC W A KEINTZ WSMR NM 88002-5502
1	USARL AMSRD ARL SL E R FLORES WSMR NM 88002-5513
	ABERDEEN PROVING GROUND
1	US ARMY DEV TEST COM CSTE DTC TT T 314 LONGS CORNER RD APG MD 21005-5055
1	US ARMY EVALUATION CTR CSTE AEC SVE R LAUGHMAN 4120 SUSQUEHANNA AVE APG MD 21005-3013
15	DIR USARL AMSRD ARL SL J BEILFUSS J FEENEY J FRANZ M STARKS P TANENBAUM AMSRD ARL SL B G MANNIX AMSRD ARL SL BA D FARENWALD M PERRY AMSRD ARL SL BD R GROTE AMSRD ARL SL BG S SNEAD AMSRD ARL SL BS W WINNER (4 CPS) AMSRD ARL SL BW L ROACH

NO. OF COPIES	ORGANIZATION
	ABERDEEN PROVING GROUND
21	DIR USARL AMSRD ARL SL BA G BRADLEY R DIBELKA M KUNKEL AMSRD ARL SL BD J COLLINS (5 CPS) A DRYSDALE T HOLDREN L MOSS E SNYDER AMSRD ARL SL BE K BATES R SAUCIER AMSRD SL BS J ANDERSON J AUTEN R BOWERS M BURDESHAW E DAVISSON V PHU G SAUERBORN

INTENTIONALLY LEFT BLANK.